Proposal for C

Array Notation for Vectorization

Javier A. Múgica

October 10th, 2025

TABLE OF CONTENTS

1 Introduction

Intent of the feature

The idea of the proposed extension is to be able to write code like

```
A[0:5] += 2;
A[:] = B[:]*C[0][:];
A[b:n] = -B[0:n];
A[0:5:2] = 1/A[0:5:2];   // Operate on the elements 0, 2, 4, 6, 8
q[0:n] = sqrt(p[0:n]);
```

On the one hand, this makes code more synthetic and easier to read than "for" loops. On the other, it helps the compiler in taking advantage of vector instructions present in the processor and, more generally, deciding which is the best way to transform that construction into machine code.

It also opens up the door for functions taking and returning arrays. This is possible because we have chosen arrays with selected elements not to decay to pointers. This is explored in "Further extensions".

This paper

This is a second version of *Array notation for vectorizarion,* N3716 (ANFV, for short). In some aspects it has been simplified, while in others it has been expanded. The main changes from the previous paper are the following:

The greatest simplification comes in the discussion, where choices that were finally not taken are not discussed at length, as they are there. In addition, range function calls are treated more naturally, in line with other range operations. The approach taken in the first version for function calls was a consequence of the evolution in the choices for the semantics of range selections and was not the right choice in the end. These range function calls are now included in the main proposal.

The possibility that zero-step selections are not supported by the implementation, already discarded in that proposal, has been completely removed from the wording. By contrast, we allow selections of length zero when the length is not given by an integer constant expression. We have also removed the `[::]` selection because it conflicts with the semantics given for this by OpenMP. Removed, too, are array casts. Functions returning arrays were considered for addition, simply by lifting the constraint forbidding them in function declarators, but in the end we did not.

We note in the wording that the type of something like `p[0:6]+q[0:n]` is a fixed-size array. We have added the "unqualified, non-atomic version" for the case the array that the range selection applies to is not an lvalue.

We give a suggestion for harmonising new sequence points (e.g., assignmnet operators) with range operations.

Some parts include wording that makes use of the "discarded" concept. These are constraints that take effect only when the expression is not discarded. Likewise, the condition "whenever it is evaluated" has been added to several "shalls" in "semantics" sections. Finally, expressions that referred to an array length as "given by an integer constant expression" have been replaced by "fixed constant".

Prior art

The specification of a range in the index or a comma-separated list of values is common in many languages. Most notably, in Fortran. Some languages specify begin and end. This is the case, for instance, of Rust slices. But Rust slices only admit compile-time constants, so in practice it would not be different from beginning and length. In C, it need be beginning and length, because the types of `A[0:5]` and `A[n:5]`, where `n` is of integer type, should be the same, namely `typeof(A[0])[5]`. This is not possible if the latter is written `A[n:n+5]`.

Vectorization with notation and semantics essentially as proposed in this document are already present in OpenMP extensions for C.

A proposal along these lines was already presented: N2081 Array sections for C, by Clark Nelson. It seems that the reason this was not pursued was the lack of any volunteer to do the work. Compared to that proposal, this one is much more detailed. That proposal introduces the term *rank,* corresponding to our *depth*. It also appears to include what we call indexed selections (that we leave out of the wording), though no example is provided. Finally, that proposal included two ad-hoc utilities: reduction operators and __sec_implicit_index. While they may be useful, they seem too specific to justify enlarging the language for their sake, and at any rate not before the main proposal is already implemented and being used.

Empty selections

When considering multidimensional selections, empty, zero-dimensional selections come out naturally. They are needed in particular for the equality (comparison) operators. They provide a natural way of making an array un-decay-able to pointer, thereby allowing assignment of arrays and functions returning an array, as in `B = A[]` and `return A[]`. This feature is by itself useful and may be implemented independent of array selections proper.

Intent of the proposal

We believe this should become a technical specification or any other form conveying a commitment from the committee. Inclusion in the standard is not possible at this point; implementation experience is obviously needed. But implementations are reluctant to include new features; once a feature is included it can never be removed (except for compilers targeting a very specific market). Another reason is that the way an extension is provided by the implementation may be different to the way it is finally adopted in the language. This makes implementations especially reluctant in regard to extensions that are likely to be incorporated to the standard. A technical specification can act as a guide for implementations, so that they can embark in the task of implementing it with confidence that they are implementing the design that may eventually be adopted by the standard. In doing this they may detect issues with the proposed wording that need adjustment, and help evolve the wording towards its final form.

A gradual specification of the feature, where a first proposal just addresses the simplest cases ignoring any possible extension, may lead to wrong decisions in the design, that reveal themselves wrong only when the feature is extended, and by then it will be too late to change.

Terminology

singleton: An object or value which is not of array type.

Note that this excludes the type **`void`**.

array's singletons: The elements, not of array type, that compose an array: If the element type of the array is not an array type, the elements of the array are its singletons. If the element type is an array type, the singletons of the array are those of its elements.

(array) length: The number of elements of an array.

known constant length: Said of an array where the length is given by an integer constant expression.

top-level variable length array: Not of known constant length

known constant total length =
known constant size: Said of an array where all the lengths are given by integer constant expressions (i.e., its length, the length of its elements if they are of array type, etc.).

There do not exist singletons of variable size, wherefore both terms coincide in practice. What is really intended when either of these terms could be used is the constant size. Therefore, we always use the second one. Furthermore, the term "total length" is never needed.

variable length array: Not of known constant size.

2 Selections [B:L], [B:L:s] and [:]

Semantics of range selection

Consider the code

```
A[0:5] += 2;
A[:] = B[:]*C[0][:];
A[:][0:6] += A[:][6:6];
```

If `A` is an array or a pointer `A[B:L]` denotes, or *selects*, the subarray of `L` elements `A[B]` ... `A[B+L-1]` and `A[:]` (possible only if `A` in an array) selects the whole array. Each of the selected elements is operated. In the second example `A`, `B` and `C[0]` have the same number of elements. In the third, for each `A[`i`]`, to its first six elements the next six are added: `A[`i`][`j`]+=A[i][`j`+6]`, for all i and for $0 \le j \le 5$.

In expressions where one of the operands is not an array with selection, that operand should be evaluated only once. Thus, in

```
int i=2;
A[0:5] = i++;
```

All five elements in `A[0:5]` are assigned the value 2 and after the expression is evaluated `i` equals 3.

The semantics for arrays with selections in just one dimension is for the most part easy with a unique obvious choice. Multidimensional selections, however, give rise to many situations which need careful decisions and wording. One of these is the combination of selections with the array subscripting operator, `[`k`]`.

Continuous subarray selection

If `A` is an array we'd like `A[B:L]` to denote the subarray of `L` elements `A[B]` ... `A[B + L−1]`. It seems this should be an array of `L` elements; that is, it should have that type. Also, `A[:]` should denote the whole array, i. e., the same object as `A`.

This poses a problem for multidimensional arrays: we'd like, e. g., `A[0:4][0:3]` to denote a bidimensional array, not the first three elements within `A[0:4]`. But if `A[0:4]` is to be an array just like `A`, differing only possibly in the number of elements, how can that be achieved? It is necessary that `A[0:4]`, in addition to its type, carries some property with it that can distinguish it from an array of four elements of the same type. We will say that `A[B:L]` or `A[:]` have elements selected, or that they are arrays with selection, or that they carry a selection, which a "plain" array does not.

Thus, after a declaration of the type

```
int A[10];
```

`A[0:3]` has type "array of three int". In addition, it has its three elements selected.

This makes possible to select the next dimension if there is one:

```
int A[4][5];
int B[10][5];
A[0:3];
B[0:4][0:3];
```

Here both `A` and `B[0:4]` have type "array of four array of five int", but `A` does not carry a selection of elements while `B[0:4]` does. For this reason

```
A[0:3]
```

denotes an array of three elements of type "array of five int", which makes 15 elements, while

```
B[0:4][0:3]
```

denotes an array of four elements, of type "array of three int" (we will justify later that the type shall be array of three int and not array of five int). Each of those four arrays has its three elements selected.

The rule is that, in an expression of the form

`A[B:L]` or `A[:]`,

if `A` does not carry a selection the operand selects elements from the first (outermost) dimension of `A`, of type that of `A[0]`, while if it does already have selected elements it selects elements from each of the already selected elements: in the expression `A[B:L]`, for each `𝔰` selected in `A`, the elements `𝔰[B:L]` are selected.

So, if `A` is an array of n dimensions, its selection, if any, will have the form of an m-dimensional array of l-dimensional arrays, and $m+l = n$. The number of, necessarily consecutive, dimensions carrying selection might be called the *depth of the selection*.

We do not allow further selections applied to an array carrying a selection whose elements are pointers. See "Further extension" for a discussion.

The type after lvalue conversion

When working with arrays with selections it is soon realised that what matters for an operation between two of them to be possible is just the number of elements selected from each dimension, not which particular elements are selected not even how many elements the full array has. For example:

```
int A[24][8][3], B[6][8][10];
A[0:6:2][0:4][:] + B[:][2:4][7:3];
```

But then we can completely discard the non-selected elements and make the type of the arrays with selections in this example be `int[6][4][3]`. There is no concern here about the memory layout of the array after lvalue conversion, since the result of such a conversion is just a value, not an object.

This is not possible for lvalues:

```
int A[18][8][3], B[6][8][10];
A[0:6:2][0:4][:] = B[:][2:4][7:3];
```

Here it is precisely the elements `A[0:0:0]`, ... `A[10][3][2]` which need have their values updated.

Broken arrays

We said above that `A[:]` or `A[B:L]` are arrays like `A`, differing possibly in the second case in the number of elements. Let us consider the case `A[B:L]` when `A` is itself of the form `C[:]` or `C[B:L]`, as in the following example:

```
int A[10][5];
A[2:4][0:3];
```

The layout of `A[2:4][0:3]` is as follows, where $\mathfrak{s}$ denotes an element that has been selected and will be operated and i an element which will be ignored:

$\mathfrak{s}, \mathfrak{s}, \mathfrak{s}, i, i, \mathfrak{s}, \mathfrak{s}, \mathfrak{s}, i, i, \mathfrak{s}, \mathfrak{s}, \mathfrak{s}, i, i, \mathfrak{s}, \mathfrak{s}, \mathfrak{s}, i, i$

The twelve elements which constitute the selection, the ones which will be operated (e.g. as `A[2:4][0:3]*=2`), are not stored consecutively. We will call one such selection, which is like an array with padding, a *broken array*. We may assign to `A[2:4][0:3]` type `int[4][3]`, or `int[4][5]`. Both options have advantages and drawbacks. The former choice will be called the *short interpretation*, while the latter will be called the *long interpretation*. There is also the possibility of assigning it a new type.

Our long experience in the elaboration of this proposal showed beyond doubt that the right choice is the short interpretation.

Ignored elements are not padding

In the long interpretation, ignored (not selected) elements seemingly act as padding when the array is operated:

```
float A[10];
A[0:4:3]=2;
```

$\{\mathfrak{s}, i, i, \mathfrak{s}, i, i, \mathfrak{s}, i, i, \mathfrak{s}\}$

Only selected elements are operated, ignored elements are not.

But there is a difference: padding bytes need not be copied onto; ignored elements *must not* be copied onto. In doing so, elements from the full array would be modified which cannot be modified. Thus, those ignored elements act as bytes outside the operated object, whose modification would change some other object: The array `A[0:4:3]` consists only of the four `s` elements. It is an array of four elements stored sparsely.

Selection from an array of pointers

As in

```
int *B[10];
B[2:4];         //Array of four pointers
B[2:4][0:3];    //[a,b,c] [a',b',c'] [a'',b'',c''] [a''',b''',c''']
```

The last object is impossible to accommodate in the long interpretation; it cannot be given a type. The extension needed to give a type to that expression is precisely the short interpretation: it is an array of four elements, even though they may be stored apart from one another in memory.

The *broken* pseudo-qualifier

We cannot just say that `A[2:4][0:3]` above has type `int[4][3]`, since the memory layout of the object is not that of an `int[4][3]`. The solution to this is to attach a qualifier to the type. Thus, `A[2:4][0:3]` is not a "plain" `int[4][3]` but a *broken* `int[4][3]`. Brokenness is a qualification the array carries, as could be its being const. The register storage class achieves a similar effect: the memory layout may be different from that of an object without that storage class. Contrary to a type declared with a qualifier, a storage class specifier does not change the type of the object.

Do we want broken arrays to be const-like (qualifier) or register-like (storage-class)? When a storage-class specifier is attached to an identifier, wherefore it applies to the object the identifier refers to, the specifier could have been omitted from the declaration (it may be necessary for the consistency of the whole program, but the declaration itself does not need it). This is not the situation for broken arrays: `A[2:4][0:3]` cannot be not-broken: it *is* broken. Storage-class specifiers refer to where the translator places (or looks for) the object and, related to it, its linkage. `typedef` is placed among them only for syntactic convenience and `auto` by accident (since C23). A broken array is a subobject of the original array, and this will be placed and have linkage according to its storage-class specifiers; selecting a subobject from it does not change this.

For this reason we prefer to make brokenness a qualification.

The type of the arrays considered above is now

```
int A[10][5];
A[2:4];         //int[4][5]
A[2:4][:];      //int[4][5]
A[2:4][0:3];    //int[broken 4][3]
A[2:4][0:3:2]; //int[4][broken 3]
```

The word *broken* is not a keyword proposed, just a symbolic way of representing the type of broken arrays. Brokenness propagates upwards: if an array is broken, so are arrays containing that array as element. In the example above, `A[2:4][0:3:2]` is a broken array, and we could also have represented its type as `int[broken 4][broken 3]`.

As already noted, the elements of the original array that were not selected do not form

part of the array after selection: they are not padding, they just are not.

Broken vs. potentially broken

Having chosen the short interpretation, broken arrays must be given a different type than plain arrays, since the memory layout is different, and, e.g., `memcpy` will not copy one onto the other. That difference is taken into account by a qualifier: *broken*. That qualifier is not introduced in the type system: no identifier can be declared with that type, no pointer to an object of that type can be formed and broken objects, i.e., expressions having as type a broken array (which are necessarily lvalues) are not allowed in **`typeof`**. Since, further, qualifiers are discarded upon lvalue conversion, omitting the mention that these arrays are broken would not change nothing in the semantics of any expression. The qualification is introduced so that properties of unqualified arrays do not change by the introduction of broken arrays.

Now, should any array with selection, or any one with a >1-dimensional selection or a `[B:L:s]` selection where `s` is not an ICE with value 1 be called "broken", or only those actually broken? Consider the expressions

```
A[0:8:s];
A[:][0:l];
```

These expressions are broken arrays or not according to the runtime value of `s` and `l` respectively. Attaching the qualifier only to those actually broken means that the type of an expression cannot be determined during translation. But the qualifier has not been introduced in the type system, as explained above, and an implementation can ignore it completely, as if it didn't exist. A translator need only keep track of the selection carried by an array, irrespective of how the standard chooses to call it. The only place it could be noticeable is as operand to **`typeof`**, and there arrays with nonempty selections are not allowed, whether broken or not. For these reasons we prefer to phrase the text in a way that only actually broken arrays, where the memory layout is not the same as that of a plain array, are deemed broken. If the qualifier were introduced in the type system and the address to broken array taken, **`broken`** `T*` should mean pointer to potentially broken array, just as **`const`** `T*` means pointer to potentially const object. Our choice leaves this possibility open for the future.

Consequences for implementations

Impelementers can ignore the broken qualifier altogether. The type of a broken array cannot escape from the expression designating the array itself. To say that this type is qualified is needed for the consistency of the type system and the standard in various parts, but the only thing that an implementation needs to care about is the memory layout; i.e., where is each of the selected elements, in order to retrieve them for the range operation of which it is part, in case it is part of such an operation.

Changes need to the wording in "Types"

And in "Representations of types". Some change is needed, since broken arrays do not conform to the descriptions given there: Unlike ordinary arrays, they are not a *contiguously* allocated set of objects and they cannot be copied in an object of its size with `memcpy`. We can think that the paragraph containing the remark on `memcpy` can be valid for broken arrays if we interpret the use of `memcpy` as an example that need not apply to all types. Arrays carrying a selection with step zero pose a problem, since they are not represented "using *n* (bytes), where *n* is the size of an object of that type". A detailed reading of that paragraph in connection with the second one, which says that "Except for bit-fields, objects are

composed of contiguous sequences of one or more bytes", reveals that the paragraph with the remark on `memcpy` is taking for granted that the object bytes are consecutive; `memcpy` is mentioned as an example amongst other ways to copy the object bytes into an array of **`unsigned char`**, and is not intended to mean a method that can be applied on some type of objects and on others no.

As we noted in the previous paragraphs, brokenness is present in the wording solely because it is formally needed. So, the least intrusive the changes in the wording, the better. Therefore, we just make broken arrays also an exception in those subclauses, note that their elements are not stored consecutively as in a plain array, and refer to the section on range selections for the description of how exactly they are laid in memory (in the abstract model).

The type of A[B:L] for nonconstant L

A variable length array

The type of `A[B:L]` is always **`typeof`**`(A[0])[L]`, possibly broken. Consider

```
A[0:n] /= 2;
```

It seems that an implementation should support this even if it does not support variable length arrays. A closer thinking reveals that the equivalent to `A[0:n]` is not an object declared as a variable length array but a "for" loop. The translator need not handle any memory allocation.

C23 does not require support for VLA of automatic storage duration. We also have from the standard that *An lvalue is an expression (with an object type other than* ***`void`****) that potentially designates an object;* So, `A[0:n]` is an lvalue and designates an object which may have (and will typically have) automatic storage duration and is a VLA. We want support for these mandatory.

The wording for making these mandatory should be in the text explaining the meaning of the **`__STDC_NO_VLA__`** macro, which states what is not mandatory. Then, the wording for the latter can take an include approach or an exclude approach. On the one hand, it may state precisely which VLAs are not mandatory. These would henceforth be the objects with automatic storage duration *declared* as VLA. On the other it may rule out which ones are not mandatory.

First approach:

`__STDC_NO_VLA__` The integer constant **`1`**, intended to indicate that the implementation does not support the declaration of variable length arrays with automatic storage duration.

Second approach:

`__STDC_NO_VLA__` The integer constant **`1`**, intended to indicate that the implementation does not support variable length arrays with automatic storage duration which are not part of an object of known constant size.

The second approach is more conceptual, focusing on the reason why those which arise from a range selection should be mandatory, not mentioning specifically range selections. If there were other instances in the language of those, they should be mandatory. There is one corner case:

```
union{int a[100]; int b[n];} u;
```

Here, `b` is part of the larger (or equal) object `u`. This should not be made mandatory. Therefore, we took the first approach.

Stepped selections

We want also to allow "stepped" selections: `A[0:5:2]`, which selects the elements `A[0]`, `A[2]`, `A[4]`, `A[6]`, `A[8]`. Or `A[0:n:2]`, which would select `A[0]`, ... `A[2(n-1)]`. The third integer in the selection is the *step,* which can be positive, negative, or zero with some restrictions, and need not be an integer constant expression.

All we have discussed applies to stepped selections, with one exception: a stepped selection as the first, outermost one. This creates from the outset an array with its first dimension broken, but this poses no problem.

Needless to say, if `s` is negative it is the highest element which is operated first:

```
A[2:3:-1]= B[0:3]; // A[2]=B[0], A[1]=B[1], A[0]=B[2]
```

```
A[8:3:-3]  --lvalue conv.-->  {A[8],A[5],A[2]}
```

s equal to zero

`s` might evaluate to zero. If the array with selection undergoes lvalue conversion, `L` copies of the element `A[B]` will fill the array. Before lvalue conversion the array has `L` elements, even if they will share the same location in memory. `s` cannot be zero if the array is the left operand of an assignment:

```
A[n:10:0]+=1; // Not allowed
```

In the proposed wording, instead of saying "if `s` evaluates to zero the array shall not be the left operand of an assignment operator" we write "an array which carries a stepped selection where the step is zero shall not be the left operand of an assignment operator". This is to make unambiguous that the following is valid:

```
A[6:10:0][2]= 2.0; // Equivalent to A[6]= 2.0.
```

It also makes valid, according to the precise definition of *stepped selection* that will be given, an `s` equal to zero provided `L` is one (or zero, see below "Selections of length zero"):

```
A[6:1:0]= 2.0; // Valid. Equivalent to A[6]= 2.0.
```

The array `A[B:L:0]` has l elements. If l is $\neq 1$, its layout is very different from a plain `int[`l`]`; its l elements share the same space in memory. For this reason its type has to be qualified somehow. We could say that it is *collapsed*, but we prefer to reuse the broken qualification for this.

Definition of broken array

Broken does not necessarily mean spread out, but *having a layout different from the unqualified array;* An array is broken if any of these happens:

— It's elements are broken.

— It's elements are not stored consecutively, with the 0^{th} first, the 1^{st} second, etc.

The kind of expressions allowed for B, L and s

We believe it should not be more than *conditional expression*. As regards side effects, our opinion alternated between prohibiting them and allowing them. Side effects would complicate the translation and be a source of bugs, as in `A[n:n++]=0` or `A[0:b[0]=3] + b[0:3]`. On the other side, these constructions have parallells in what can currently be done:

```
n= n++;                (b[0]=3) + b[0]
```

As for the complication in the translation, side effects within the brackets are unsequenced with respect to the range operation. This actually makes translation easier, for the translator can choose the point at which side effects take place, though not useful, since the result is not well defined. This also has parallells in current code: `A[A[0]++]` may evaluate to `A[A[0]]` or to `A[A[0]+1]` or, in theory, to `A[`*i*`]` for any *i*, if the value of `A[0]` is being updated while being read as the subscript for the outermost `A`, or to any value whatsoever if the the outermost `A` wants to read `A[0]`. Finally, there are use cases that seem natural, as

```
A[0:n++]= x;       A[k:3] = B[f(k):3];
```

So in the end we allow them. We will say more about the order of evaluation in “sequence points” below.

Conditional expressions cause no ambiguity in the interpretation of `:`. Parentheses can ease readability: `A[(b ? 0:n) : n]`.

Range operation

Whenever an expression is of the form `A[R]`... *op* `B[R’]`... , where *op* is an operand that operates the element to its left with the element to its right and `[R]` and `[R’]` are range selections, the number of selected elements in `[R]` and `[R’]` have to be the same, call it *n*, and the expression is equivalent to

$$\texttt{A[}s_i\texttt{]}\ldots\ op\ \texttt{B[}s_i\texttt{]}\ldots, \quad 0 \le i < n$$

`A[`s_i`]` and `B[`s_i`]` run through the selected elements. Similarly if the operation is of the form `A[R]`... *op* `s`, where `s` is a singleton. We will call this a *range operation,* say that *op* acts as a range operator and that `A[R]` and `B[R’]` are *treated as a range of objects*.

If the elements that are operated are `A[]` *op* `B[]`; this is, two matrices that do not decay to pointers, either because they appear us such in the code or because they are each of the operations into which a range operation decomposes, the elements are usually operated one by one, as thought it were `A[:]` *op* `B[:]`, and we still call this a range operation.

Restrictions on the values of B, L and s

In general, the restrictions pursue that the indices $b+i*s$, with $0 \le i < l$, fall within bounds. But there are exceptions: If the expressions are integer constant expressions, the restrictions applies only when the range selection is not discarded. If they are not constant, they apply only if (and each time that) the expression is evaluated; this cannot be otherwise, for it doesn’t make sense to impose any requirement on the value of an expression that is not evaluated. The ICE case goes to “constraints” sections, while the other case is treated in “semantics” sections.

This dichotomy of ICE and not ICE causes some duplication of conditions, placed as a

constraint for ICE and repeated in the "semantics" section for non-ICE. Not all of them are duplicated, however. Some "absurd" values that are forbidden if given by an ICE are allowed if the expression is not an ICE, provided it will not cause an out-of-bounds access. For example, a negative value for *b* is allowed provided *l* is zero, the latter in turn only allowed if `L` is not an integer constant expression. Likewise, a step zero at the left hand side of an assignment expression is outright forbidden only if given by an ICE.

Not all restrictions placed on ICE values are turned off in discarded contexts. When two array lengths in the same expressions need be the same, this is enforced also in discarded code, because otherwise the type of the expression would not be well defined:

```
sizeof(A[0:6]+B[n;7])
```

Array subscripting applied to an array with selection

Consider the following:

```
int A[9][6];
A[5:4][0];
A[5:4][1:2][0];
```

The following appealing interpretation, in which the `[`*k*`]` operand applies to each of the selected elements:

```
A[5:4][0];  // {A[5][0], i, i, i, A[6][0], i, ... A[8][0]}
A[5:4][1:2][0]; //Wrong, there are not three dimensions
```

is at odds with the definition of the `[`*k*`]` operation as selecting the *k*-th element of the object to its left. `A[5:4][0]` should be `A[5]`. This means that for the `[`*k*`]` operator the array of selected objects should not be treated as a range of objects. The semantics of the above examples should be:

```
A[5:4];         // {A[5][0], ... A[5][5], A[6][0], ... A[8][5]}
A[5:4][0];      // A[5] = {A[5][0], A[5][1], ... A[5][5]}
A[5:4][1:2][0] // {A[5][1], A[5][2]}
```

If the user wants to select the `0`-th element from each selected array, he should write

```
A[5:4][0:1];
```

Stepped selections result in holes between the selected elements For example, the layout of `A[10:4:3]` is

{𝓈, *i*, *i*, 𝓈, *i*, *i*, 𝓈, *i*, *i*, 𝓈},

where *i* stands for "ignored" and 𝓈 for "selected".

Any array `A[10:4:s]`, where `s` may not be an ICE, can be represented as {𝓈, 𝓈, 𝓈, 𝓈}, where it is *not* understood that the elements are consecutive in memory. Its elements will be denoted by `[0]` ... `[3]`. `A[10]` is the one denoted by `[0]`, whether *s* be positive or negative. In general, in a selection `A[B:L:s]`, the *i*-th element, `A[B:L:s][`*i*`]`, is the one at `A` + *b* + (*i*-1)**s*, and 0 ≤ *i* < *l*.

Note in passing, that if such a broken array is part of an array, saying for the enclosing array that its elements are or not stored consecutively doesn't make sense. For example, consider `int A[5][10]` and `int B[5][20]`, and the selections `A[0:2:2][0:3:2]` and

`B[0:2][0:3:2]`, that have identical layouts. Can we say that the two elements of `A` are not stored consecutively but those of `B` are?

All the above applies before lvalue conversion. After lvalue conversion there is nothing to say about its memory layout, for it is, conceptually, just a value, an abstract array with its singletons having values of a certain type.

Pointer to one of its elements; valid offsets

We forbid taking the address of an array with nonempty selection. But the address of one of its elements may be taken and the user can make use of that pointer to access elements from the array:

```
int A[10], B[6][6];
int *p= &A[0:4:3][0];
int *q= &B[:][0:3][0][0];
```

The arithmetic for these pointers is the same as that of any other pointer to **`int`** (it cannot be otherwise). As with any other pointer to an array element, defined behavior is only ensured if the pointer is used to access elements from the array it was extracted from, not from any larger array of which the former array is a subarray:

```
p[0]=0, p[3]=1;      // Valid
p[1]=2; p[0]=p[4]; // Invalid (U.B.)
q[2]=q[1]=q[0]=4;   // Valid
q[4]=0;              // Invalid
```

This restriction allows the compiler to reason about elements not modified by the pointer:

```
int A[6][3], B[6][6];
int *q= &B[:][0:3][0][0];
/* Operations using q */
A[:][:] = B[:][3:3];
```

In the code above the compiler can reason that no access through `q` modifies the upper three elements of each `B[`*i*`]`, and advance the reading of those elements, needed for the last instruction (e.g., by placing the whole instruction before others using `q` in the generated code). Here is another example:

```
int A[6], B[6];
int *q= &A[3:6:0][2];   // q is &A[3]
q[0]=5;   // Modifies A[3]
q[1]=1;   // Invalid
```

The element of which the address is taken can itself be an array, but not an array with selection:

```
int A[6][6];
int (*p)[6]= &A[0:4:3][0];
p= &A[:][:][0];   // Invalid: address of an array with selection
```

0-dimensional selection

After the declaration `int A[3][3]` we have the following possibilities with respect to the dimensions of the selection (its depth) and that of the selected elements, to which we add now the last entry:

Selection	Dim. of selection	Dim. of selected elems.
`A[:][:]`	2	0
`A[:]`	1	1
`A[]`	0	2

The last option selects the matrix as a whole. It has one selected element, which is a 3×3 matrix. This is the formal explanation. In practice for the programmer it means that the matrix is treated as a whole, for instance for the `==` and `!=` operators, and that it does not decay to a pointer, the latter formally because it carries a selection. An empty selection is a selection of depth zero.

Once we have settled onto providing some way to perform a 0-dim. selection, a syntax has to be chosen for it. We considered two options:

`[A]` `A[]`

None of them presents incompatibilities or ambiguities with the current uses of the `[]` operator. The first one, since for its use as array subscripting an expression must precede it, while in the construction `[A]` here, an expression cannot precede it. For the second one, an empty `[]` is only allowed in declarators, and even there only in some places.

The second one is in keeping with the syntax of the other range selections. On the other hand, the semantics of a zero dimensional selection is essentially to avoid the matrix decaying to a pointer. It has no effect when applied to a matrix that already carries a selection:

`A[0:5][][:]` is equivalent to `A[0:5][:]`

and it wouldn't be needed for the `==` operator if its operands did not decay to pointers. With this view, the syntax `[A]` conveys the idea that the brackets *protect* the matrix; then we could say that a *protected* matrix is never converted to a pointer. It also becomes clearer that this *matrix-protection* operator has no effect on an array with selection, which cannot already decay to a pointer:

`[A[0:5]]` is equivalent to `A[0:5]`

If the protection of a matrix is part of the replacement text of a macro, this may lead to the occurrence of two consecutive `[` tokens: `[[A]]`, which has the syntax of an attribute. This can be solved by defining the macro as

```
#define protect(a) [(a)]
```

The other obvious choice, `[ a ]`, is not possible because of the decision of C of interpreting `[[` not as one token but as the succession of two tokens.

A more important reason strikes a death blow on this syntax: An array within brackets may be needed, and will very likely be in some implementation providing the extension, for specifying an arbitrary sequence of indices to be selected:

```
unsigned int I[3] = {0,4,3};
float B[3], A[10];
B[:] += A[I];
```

Strictly speaking, this does not rule out the `[A]` syntax for a zero selection, since in one case the operator must be preceded by an expression of array or pointer type and in the other case it cannot. But using the same syntax for two different operations with the same possible type for the operand is a very bad choice if it can be avoided.

Therefore, we chose the `A[]` syntax over the `[A]` alternative.

Equality operators

What should `A[:] == B[:]` yield? From the point of view of the `==` operator it should evaluate to true if all elements are equal. From the point of view of a range operation it should yield an array of `0`'s and `1`'s. It will eventually become unavoidable to provide syntax for the two options. If either or the other choice is taken for the equality operator, it seems that another operator would be necessary for the other.

Let us analyse the range operator point of view:

```
int A[3][3], B[3][3], C[3][3], D[3], E;
C[:][:] = (A[:][:] == B[:][:]);
D[:] = (A[:] == B[:]);
E = (A[] == B[]);
```

Making `==` operate as any other operator on the range of selected values lets the programmer choose at what level the comparison is carried: At each singleton level, resulting, in the example, in a 3×3 matrix; at the level of the next to last dimension, comparing vectors, the result being true if all components are equal, yielding the value assigned to `D` in the example, which is equivalent to

```
D[0] = A[0][0]==B[0][0] && A[0][1]==B[0][1] && A[0][2]==B[0][2];
D[1] = A[1][0]==B[1][0] && A[1][1]==B[1][1] && A[1][2]==B[1][2];
D[2] = A[2][0]==B[2][0] && A[2][1]==B[2][1] && A[2][2]==B[2][2];
```

Or comparing all the elements yielding a single value, assigned to `E` in the example. Thus there is no need for two different operators, just syntax for a 0-dimensional selection.

Therefore, the right choice is to treat `==` and `!=` as any other binary operator and operate on the selected elements.

We define `A != B` to be `!(A == B)`. Here, the result of `A == B` is a matrix with all its singletons selected, and `!` negates each of them. For example, with `A` and `B` as above,

`A[:] != B[:]` is equivalent to `!(A[:] == B[:])`

`A[:] == B[:]` yields an array of three elements with all its elements selected, and the `!` operator negates each of them. That is, the result is equivalent to

```
[0] = !( A[0][0]==B[0][0] && A[0][1]==B[0][1] && A[0][2]==B[0][2] )
[1] = !( A[1][0]==B[1][0] && A[1][1]==B[1][1] && A[1][2]==B[1][2] )
[2] = !( A[2][0]==B[2][0] && A[2][1]==B[2][1] && A[2][2]==B[2][2] )
```

Relational operators

The relational operators are `<`, `>`, `<=` and `>=`. When a relational operator is applied to an array with selection we mandate that the selected elements be singletons. This excludes the analogous to `A[:] == B[:]` or `A[] == B[]` in the examples above. While it is clear that two vectors or matrices should compare equal only if all elements compare equal, it is by no means clear that `A` should be `< B` if all elements of `A` are less than all corresponding elements of `B`. Furthermore, this would make `<` and `>` not the opposite of `>=` and `<=` respectively.

That semantics can be achieved with the current proposal, though an intermediate variable is needed:

```
C[:][:] = (A[:][:] < B[:][:]);
(C[] == 1)
```

The equivalence of matrix without or with 0-dim. selection

We saw that an empty `[]` selection is sometimes needed for preventing the matrix to decay to a pointer (we recall, we called this to *protect* the matrix). Both:

`E[] = D[];` and `E = D[];`

are possible and have the same meaning.

`E[] = D;` and `E = D;`

are not possible, because `D` decays to a pointer (but they still have the same meaning).

For passing the matrix as argument to a function, in the event that function arguments are extended to allow arrays, the 0-selection is the most natural syntax;

```
double determinant3(double A[:3][3]);
determinant3(A[]);
```

It is also the most natural option when we want the type of a matrix to be that of an identifier declared with inferred type:

```
auto a = A[];      auto a = A[:];      auto a = A[::];
```

All three forms are equivalent, but it seems that the first one conveys meaning better, since the selection is irrelevant.

In the three examples, the `[]` operator has no other function than to protect the matrix. We'd like that to be the general case; that is, there should be no semantic difference between a matrix which does not decay to a pointer and a matrix with a zero selection. If we want to preserve this in the current proposal we may need to allow matrices with 0-dimensional selection in places where we will preclude matrices with selection. This will be treated in the next section. It turns out there are only two such situations:

`&A[]` `typeof(A[])`

We forbid taking the address of an array with selection or placing it as the argument of **`typeof`** because it can be problematic for broken arrays. There is no problem therefore in allowing arrays with zero dimensional selections there. It should be noted that this does not make a selection like `A[B:L][]` allowed as operand for those operators, which would

completely break our decision of not to allow arrays with selections in those places, because a `[]` selection applied to an array which already carries a selection has no effect:

```
int A[5][5];
&A[]; &A[0][]; //Allowed
&A[0:1][]; //Not allowed. A[0:1][] carries a nonempty selection.
```

4 Restrictions on arrays with selection

Inconvertibility to pointer

As already noted and assumed in the previous sections, an array with selection is not convertible to a pointer. Far from being a defect we see this as an advantage, for it opens the door to arrays being used in many places it is currently impossible; some of these have already arisen, others will come up later.

Restrictions for broken arrays before lvalue conversion

The type and layout of an array before lvalue conversion is needed in particular in these contexts:

typeof operators **sizeof** **&** operator

sizeof

```
int B[10][10];
sizeof(B[2:3]);          // sizeof(int[3][10])
sizeof(B[:][0:5])        // sizeof(int[10][5])
sizeof(B[2:3][0:5][0]) // sizeof(int[5])
```

The value "returned" by the **sizeof** operator is the same before and after lvalue conversion. Therefore, we allow arrays with selection as operands to **sizeof** unconditionally. The size of `A[B:L]` and `A[B:L:s]` is always *l*`*sizeof(A[0])`, where *l* is the value to which `L` evaluates.

If *s* is zero, the size in memory is `sizeof(A[0])`, not *l*`*sizeof(A[0])`. We prefer to keep the value which **sizeof** yields as *l*`*sizeof(A[0])`. This means that for these arrays **sizeof** does no return the number of bytes the object takes in memory. But returning this latter value would constitute an exception to the rule *l*`*sizeof(A[0])`, i.e., `_Countof(A)*sizeof(A[0])`, and this would bring in many problems: The value given by **sizeof** would not be the size needed to store the array *value* in some other object; `sizeof(A[B:L:s])` would no longer be an ICE whenever `L` is, but both `s` and `L` need be integer constant expressions, **sizeof** would not return the same value for all objects of type *T*`[`*l*`]`, and **sizeof** would return different values before and after lvalue conversion of its argument.

& operator

A pointer to a broken array addresses an object with a different memory layout as a pointer to a plain array. Therefore, it must carry the broken qualifier; i.e., it is a pointer to a broken array. This is consistent with the way qualifiers work.

The pointer needs to remember the memory layout of the array it points to; i.e., the exact way the array is broken. This information on the layout is copied with the value:

```
int B[10][8], C[6][4];
int (*p)[broken 5][2], (*q)[broken 5][2];
typeof(p) p2; typeof(q) q2;
p= &B[0:5][0:2];
q= &C[0:5][0:2];
&p[1][0] - &p[0][0]; // 8
&q[1][0] - &q[0][0]; // 4
p=q;
&p[1][0] - &p[0][0]; // 4
```

Thus, allowing the `&` operator on broken arrays would require the enlargement of the type system and the emergence of fat pointers, and not of the simplest kind; brokenness would no longer be an ephemeral property of some arrays with selection with no impact in practice on the programmer or even on implementers. Consider for instance function parameters: A parameter declared as `int (*p)[broken 2]` does not determine the layout of `*p`. The compiler would need at runtime the hidden data carried with a broken pointer in order to compute the address of `(*p)[1]`.

This is way more than the present proposal intends. Therefore, we disallow the `&` operator in arrays carrying a range (i.e., nonempty) selection.

typeof

Since we do not want to extend the type system we should not allow `typeof` on broken arrays, or if allowed the `broken` pseudo-qualifier should be dropped. The latter option would make `typeof` inconsistent in the event that `broken` is fully integrated into the type system, allowing pointers to broken objects to be declared and constructed. We could still allow `typeof` when applied to unbroken arrays that are known to be unbroken during the translation of the expression (i.e., the brokenness or not of which only depends on ICEs). But we prefer the simpler specification that no array with nonempty selection is allowed as operand to `typeof`.

Implementation experience will show whether implementations choose to allow arrays with selection in `typeof_unqual` or are reluctant to it, informing the eventual standardization. For the time being, we allow it.

Other Restrictions

An array with selection cannot be the argument of a function call. This is not something we impose but a consequence of the fact that those arrays do not decay to pointers, and functions cannot take arguments of array type.

In places where the object is needed for its type, an array with selection is strange: the intent of range selections is to select several elements from the array, to be operated, not to place the array inside `typeof()`, say. More generally, whenever the array with selection would be placed in a position where the action would not be to operate individually on the selected elements, i.e., where the operation would not be a range operation, its use is questionable. We have identified these places:

Argument to function

_Generic, controlling expression

auto, initializer for a type inferred declaration

_Countof

alignof

unary *

in addition to **[*k*], typeof, sizeof,** and **&** treated above, and **casts** treated below. Of these, we have chosen to allow **_Generic, auto, _Countof** and **alignof**, though the latter case cannot arise.

Argument to function

The selection causes the function to act as a range operator, being called several times, each time taking one of the selected elements, as will be explained below. Therefore, an array with a range selection can be written as part of the argument list of a function-call expression, but will never be the argument of an actual function call.

_Generic

The controlling type of a **_Generic** selection is taken from its controlling expression after lvalue conversion, so there is no problem with respect to brokenness. Further, since arrays with selection do not decay to pointers, they are a way of making the controlling type an array type, which is not possible with arrays not carrying a selection.

There is no need to perform any change in the wording for this operator.

auto

The situation is similar as for **_Generic**: the type is taken from the initializer after lvalue conversion. Here we are interested in allowing arrays with selection, for it makes sense that it is the number of selected elements what we are interested in, and because it is the only way of declaring an identifier with array inferred type:

```
int A[10][6];
auto p = A;             // int(*)[6]
auto a = A[];           // int[10][6]
auto b = A[:];          // int[10][6]
auto c = A[0:5][3:3];   // int[5][3]
```

alignof

alignof can only be that of the corresponding full array, and in any case this operator only takes a type name, not an expression.

unary *

This is impossible, since arrays with selection are no convertible to pointers.

5 Casts

Casts apply to values, not to lvalues. This simplifies the design. At present the type name of a cast cannot specify an array type. This is because it cannot possibly apply to an array, which would be the only type that could be meaningfully cast to an array type. An array is allowed as the operand, but it decays to a pointer. As we now have arrays that do not decay, these should be allowed to be cast to arrays. The resulting value also carries a selection (and cannot be converted to a pointer, but this already follows from it being a value, not an lvaue). However, in the end with did not include these array casts in the proposed wording.

Changing the singleton type

For example:

```
A[:] = (float)B[:];
```

A programmer writing this expresses the intent that he wants each value of `B` to be converted to **`float`**. This per-element conversion avoids the copy required by casting the whole array. For this to be possible we have to make the cast operation a range operation. At first this seemed strange, but considering use cases we realised that not only it is useful but that it is going to be by far the most common use of array casts:

```
int *A;
unsigned int *s;
A[0:n] += (int)s[0:n];
```

for example.

By making a cast a range operation it gains in expressivity, just as we did for the `==` operator. Unlike for this operator, we allow the array to which it applies to carry only the two extreme kinds of selection: either its selected elements are singletons or it carries an empty selection. The intermediate cases appear of very limited use. We refer to those two casts as the *range cast* and the *array cast*. If the selected elements are singletons (*range cast*), the type of the cast can be anyone that is allowed for the singleton.

The compiler need not store the resulting array in memory. For example, if `B` is of type **`double`**`[4][4]`, the expression

```
(float)B[:][:];
```

has type **`float`**`[4][4]` and carries a selection of singletons. This result will be used in one of the following ways: ignored; in a place where the value of the singletons is not needed (e.g., in **`sizeof`**), or as an operand of a range operation, where each element will be operated at a time (or in groups of four to take advantage of vector instructions, say), and the compiler knows it can convert the values one by one as they are being operated.

Array cast

(As noted, these were finally left out)

Other than to its own type, an array may be cast to a different array of the same singleton type and less or equal total size:

```
float A[4][4];
```

```
(float[2][4])A[];
```

This allows a multidimensional array to be treated as a vector, as in the following example for bidimensional arrays:

```
#define len2(x) _Countof(x)*_Countof((x)[0])
#define VEC(x) ((typeof((x)[0][0])[len2(x)]) (x)[]);

float A[n][3][k], (*B)[k];
#define Ã VEC(A)
for(int i=0; i<3*n; i++) B[i][:]=i*Ã[i][:];
```

Note however that the reverse assignments are not possible: the result of a cast is not an lvalue.

The casts of this type that make more sense are the ones collapsing several dimensions into one, splitting one dimension into several ones or restricting the outermost length. But we do not see why other combinations should be prohibited, as long as the size of the target type is ≤ the size of the operand. The compiler may warn upon conversions that break the dimension layout:

```
int A[6][6];
(int[7][5])A[];     //Possible warning
(int[3][12])A[];    //Possible warning
(int[15])A[];       //Possible warning
(int[18])A[];
(int[2][3][6])A[];
```

Variants of the element type in the cast

In an array cast we allow (that is, if we included array casts in the wording) casts to an array type with a compatible type for the singletons. We also allow any qualified or atomic version of the type; as for any other cast; these qualifiers and being atomic are lost in the cast.

6 Assignments

Assigning an array

We allow the following:

```
int A[3][4], B[4];
A[:] = B[]; //Equivalent to A[0][:]=B[:], A[1][:]=B[:], etc.
```

If the left operand carries a selection where the selected elements are arrays (or if it carries no selection, for a left operand in an assignment does not decay to a pointer), the right operand shall be an array with a selection matching the dimensions of those selected elements. Here follow some examples:

```
int C[2][3][4] A[3][4], B[4];
```

```
A[:] = B[];            //Allowed
A[:] = B[:];           //Not allowed
C[:] = A[];            //Allowed
A[] = C[0][];          //Allowed. Can also be written A = C[0][]
C[:] = A[:];           //Not allowed
C[:] = A[:][:];        //Not allowed
C[:][:] = A[0][];      //Allowed
C[:][:] = A[0][:];     //Not allowed
C[:][:] = A[0][:]+B[:];        //Not allowed
C[0][0][:] = A[0][:]+B[:];     //Allowed, singletons
C[:][:] = C[0][0][];           //Not allowed, for different reason
```

If we want to copy the result of `A[0][:]+B[:]` into each `C[`*i*`][`*j*`]` we need an intermediate variable:

```
int D[4];
D[:] = A[0][:]+B[:];
C[:][:] = D[];
```

The effect of the intermediate variable `D` is to forget the selection; the selection of `A[0][:]+B[:]`, which is a selection of singletons. Writing

```
C[:][:] = (A[0][:]+B[:])[];
```

does not solve the issue because `(A[0][:]+B[:])[]` carries the same selection as `A[0][:]+B[:]`; namely, a selection of singletons.

Here are other examples, where the right array carries a range selection:

```
int A[3][4][5], C[3][5], D[3][5];
A[:][:] = C[:];
C[:] = D[:];    // C[0] = D[0], etc. Equivalent to C[:][:] = D[:][:];
```

That this is to be allowed follows from the general rule that `A[:]`... *op* `B[:]`... means `A[`*i*`]`... *op* `B[`*i*`]`..., $0 \le i <$ `_Countof(A)`. Hence, in this example, `A[0][:]=C[0]`, etc. (more formally, `A[0][:]=C[0][]`; i.e., `C[0]` does not decay to a pointer).

The reason we do not allow the ones we do not allow is twofold: In the first place, code using those expressions can be very confusing to read and understand. It is not clear whether the right array is being assigned to each selected element (of array type) at the left, as the not allowed `A[:] = B[:]` above, or whether it is a per-element assignment as in `C[:] = D[:]` here, which is allowed. Secondly, allowing those would require breaking the rule for `A[:]`... *op* `B[:]`... For example,

```
int A[3][4], B[4];
A[:] = B[:];           //Not allowed
```

should be, according to that rule,

`A[0] = B[0]`, etc.

which is not possible.

For similar reasons, the gray assignments on the right column are not allowed. The left column displays the right way to write them:

```
int A[3][3], B[3][3], C[3][3], D[3];

C[:][:] = A[:][:] - B[:][:];    C = A[:][:] - B[:][:];
D[:] = (A[:] == B[:]);          D = (A[:] == B[:]);
D[] = A[0][];                   D = A[0][];
```

Overlapping in assignment

We do not allow expressions like

```
A[0:8] = A[1:8];
A[0:5] = A[4:5:-1];
A[0:8] = A[0:8]*A[3];
A[0:5] = A[0:5] == A[5:5]; //Assignment of a single value into five places.
```

The text on assignment expressions already includes the following requisite:

If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the two objects shall occupy exactly the same storage and shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.

This makes undefined the first of the following assignments, but no the second:

```
union {int i; short j} a;
a.i = a.j;
a.i = a.j + 0;
```

In the second assignment, the value stored in `a.i` is not read from an *object,* but is the result of the expression `a.j+0`.

For range operations we need be stricter, in order to make possible for those operations to be translated into vector operations in machine code. For example,

```
A[0:8] = A[8:8]*B[0:8];
A[0:8] = A[1:8]*B[0:8];
```

In the second assignment, the vector instructions may not produce what is written.

For this reason we require that no element which is written to is read at the right of the assignment operator, except for the computation of itself. Thus, the first two assignments below are allowed but the next two are not:

```
A[0:8] = A[0:8]*A[0:8];
A[0:8] = A[0:8]*A[8:8];
A[0:8] = A[1:8]*A[1:8];
A[0:8] *= A[3];
```

The condition that an element not be read has to be understood in the abstract machine. For example, in

```
A[0:8] = B[0:8] + 0*A[1:8];
```

the implementation may choose not to read `A[1:8]`, but in the abstract machine it is read and the behavior is undefined.

Whether some “forbidden” element is read at the right might depend on input, or on the code of a function unknown to the translator, as in

```
A[0:8] = f(A);
```

The wording of the condition needs no adjustment. If, during execution, no forbidden element is read, the behavior is defined; otherwise, it is not. If, had the program been translated according to the abstract machine, some forbidden element would have been read, but it is not according to how the program was actually translated, then the behaviour the program exhibits is right under any possible interpretation. It may happen that the elements read depend on some input that in turn depends on how the program is exactly translated, but that is already the case for constructions existing in the language:

```
p[3]++ + f(p)
```

Here `f` may end up reading some or other element from the array pointed to by `p` depending on some input.

Overlapping in the range

We could also make undefined any overlapping in the range expressions with the array being assigned to:

```
A[0:A[0]] = 6;
```

The translator must evaluate `L` and `B` (as in `A[B:L]`) before translating the assignment, so the construction does not seem problematic. The situation gets more complicated if `A[0]` is also used at the right:

```
A[0:A[0]] = A[0:4]*2;
```

For this instruction to have defined behavior `A[0]` has to be `4` before the assignment, which means that it will be eight after the assignment, but there is again no ambiguity.

In the previous examples, the program will exhibit a different behaviour if the translator translates it as follows: First compute all values of `B`’s present in the instruction; then compute and assign the first value in the range assignment (which does not depend on the values of `L` or `s`); then evaluate the expressions `L`’s and `s`’s and compute the rest of the assignment.

We don’t think the previous is a desirable behaviour. We therefore would mandate evaluation of `B`, `L` and `s` before the range selection and don’t restrict them as regards overlapping with the object being assigned to. Actually, there is no wording needed for this; the standard already includes

> *The value computations of the operands of an operator are sequenced before the value computation of the result of the operation.*

The value of `A[`*b*`]` is part of the result of the operation `A[B:L]` or `A[B:L:s]`. Hence, the translator is required to evaluate `B`, `L` and `s`, which are operands, before evaluating `A[`*b*`]`.

There remains the unspecification of whether the left or the right operand is evaluated first, and this may cause undefined behaviour because of side effects in the expressions in the range, just as for any other assignment instruction.

There is no need to extend the restriction on read elements at the right also to written elements. This will happen if the right hand operand produces the side effect of modifying some element assigned to by the assignment:

```
A[0:4] = A[0:4]++;
```

Expressions like this one already have undefined behavior because of two unsequenced modifications to the same object.

7 General rules for operating arrays with selection

The general rules

The following assignments are all allowed:

```
float A[4][6], B[4][6], C[4], f;
A[:][:] *= f;
A[:][:] += B[:][:];
A[:][:] *= C[:];
```

In the first of the assignments above we have a matrix carrying a selection of singletons operated with a singleton. In the second one, two matrices carrying corresponding selections of singletons. In the third one, two matrices carrying selections of singletons but not matching.

In a range operation, when both operands are arrays carrying a nonempty selection, each pair of corresponding selected elements is operated, taking care of the fact that if these elements are arrays they cannot be converted to pointers, even if they carry no further selection.

When one operand carries a nonempty selection and the other operand is a singleton, each selected element is operated with the singleton.

When these rules are applied to the expression `A[:][:] += B[:][:]`, this implies that each `A[`i`]` is operated with each `B[`i`]`, i.e., `A[`i`][:] += B[`i`][:]`. Since these `A[`i`]` and `B[`i`]` still carry selections, the rule applies again and we get that each `A[`i`][`j`]` is operated with each `B[`i`][`j`]`.

When the rules are applied to the expression `A[:][:] *= C[:]`, we get again that each `A[`i`]` is operated with each `C[`i`]`. Now the first ones still carry a selection but the `C[`i`]` are singletons, and `A[`i`][:] *= C[`i`]` means `A[`i`][`j`] *= C[`i`]`; that is, each row of the matrix `A` is multiplied by the corresponding element in the vector `C`.

Application of the previous rules when the selections are of different depth eventually leads to an expression `a` *op* `b` where one (and only one) of the following holds:

1. both elements are singletons.
2. either `a` or `b` is a singleton and the other one is an array with no selection (or an empty one).

3.a. either `a` or `b` still carries a nonempty selection and the other one is a matrix carrying no selection or an empty one.

3.b. both `a` and `b` are matrices with no (or empty) selection.

The first case will be the most common one and needs no wording. It is the one arising in the three compound assignments above. The second possibility is generally not allowed, and no wording will be provided for it in general. In case 3a, a rule will be stated according to which the innermost selected elements of the array that still carries a selection shall be arrays matching the dimensions of the other operand, and each selected element will be op-

erated with that other operand, as if it were case 3b, and this operation is done singleton per singleton. Thus, if `a` is the operand that still carries a selection in case 3a, and `A` represents each innermost selected element from `a`, or if we are in case 3b and `A` is `a`, `A[::]` *op* `b[::]` is performed. Here `[::]` represents `[:][:]`... , as many as the number of dimensions of the matrices.

Thus, expressions that eventually lead to cases 1 and 3 are treated in the same way: each innermost selected element of one operand (it may be the whole matrix if the selection is empty), or the operand itself if it carries no selection, is operated with the analogous elements of the other operand, and the dimensions of one and the other must match (singletons in 1, matrices in 3).

Assignment, equality and relational operators

With respect to the case 3a, simple assignments (but not compound assignments) are a partial exception in that the array which remains with selection must be the left operand. Thus,

```
float A[4][6], C[6];
A[:] = C[]; //Allowed
C[] = A[:]; //Not allowed
```

This will be taken into account when listing the possibilities for the left and right operands of assignment operators, so that the general rule can be stated with no exception.

Equality operators constitute an exception for cases 3 in that the result is not as defined by the general rule, but is a single element for each `A` *op* `b`, as we have seen.

Equality operators allow the second case: `A` *op* `s` where `A` is a matrix with no selection (or an empty one) and `s` is a singleton We have seen what its semantics is.

For relational operators the third case is forbidden.

Application to multiplications of matrices

The rule for the third case makes possible the multiplication of a matrix by columns:

```
float A[4][6], B[4], C[6];
A[:][:] *= B[:]; // Multiplication by rows
A[:] *= C[]; // Multiplication by columns
```

Multiplication of a row from a matrix by a column from another matrix doesn't involve case 1 nor 3, and the way to accomplish it is not trivial:

```
float a=0, B[m][k], C[k][n];
a += B[3][:]*C[:][4:1];
```

In this example the general rule applies first to `B[3]` and `C`, yielding `B[3][`i`]*C[`i`][4:1]`. Here the general rule applies again (each `B[3][`i`]` is a singleton), the result being

$$\texttt{B[3][}i\texttt{]*C[}i\texttt{][}j\texttt{]}, \quad 0 \le i < \mathrm{k}, \quad 4 \le j < 5\,;$$

i.e., $\texttt{B[3][}i\texttt{]*C[}i\texttt{][4]}, \quad 0 \le i < \mathrm{k}$

This is an array with a bidimensional selection. This should be operated with the singleton `a`, the general being applied twice and the sum of the values stored in `a`. However, in an assignment instruction the singleton cannot be the left operand. Even if the rules for equality operators allowed it, it would result in an unspecified value for `a` due to the differ-

ent individual assignments being unsequence.

In general, performing the scalar product of two vectors is not possible, for the aforesaid reason. Since there is just one obvious semantics for *s* `+=` A[:], where *s* is a singleton and A[:] represents an array carrying a selection of singletons, that could be easily defined in the wording. A compound assignment of this sort would be specified as a sequence of *indeterminately sequenced,* as opposed to unsequenced, singleton assignments. This is the case for function calls, as will be seen. Range compound assignments and range function calls share that each of the individual instructions may have side effects on the same object (range `++` and `--` decompose to individual instructions with side effects on different objects).

Multiplying element by element two vectors is the prototype example of range operation. However, performing such a multiplication between a row from a matrix and a column from other matrix is not not easy. The following would not work:

```
float A[k], B[m][k], C[k][n];
A[:] = B[3][:]*C[:][4];
```

Because `C[:][4]` is simply `C[4]`. We need an intermediate bidimensional array of dimensions `[k][1]` or a pointer-cast hack:

```
float A[k], B[m][k], C[k][n];
*((float(*)[k][1])&A)[:][:] = B[3][:]*C[:][4:1];
```

In the previous examples, the fact that `B[3]` is part of a matrix is irrelevant. The only thing that matters is that it is a vector (a one-dimensional array).

If range selections serve to make the code neater, the pointer-cast hack instruction above is a counter-example to it. Array selections is a tool that should be used, not abused. Even if some mechanism for operating a matrix in transposed order were available, it might be better to write the multiplication of a row vector and a matrix with the combination of a “for” loop and a range operation:

```
unsigned int k,n;
float *A;                    // A[k]
const float *B, (*C)[n];     // B[k], C[k][n]

memset(A,0,sizeof(float)*k);
for(unsigned int i=0; i<k; i++) A[:] += B[i]*C[i][:];
```

Multiplication by a column vector is easier:

```
float A[4], B[4][6], C[6];
A[:] = B[:]*C[];
```

8 Other

The decaying of plain arrays to pointers

The subsection of the standard on additive operators includes the following constraint:

> *For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to a complete object type and the other shall have integer type.*

So, the operand cannot be an array, which we know it can. The text at **6.3.3.1** on conversions of other operands says that

> *An expression that has type "array of* type*" is converted to an expression with type "pointer to* type*" except when it is the operand of* [...]. *The resulting pointer points to the initial element of the array object and is not an lvalue.*

But it is dubious that this text combined with the restriction allows an array as the operand of the additive operator. It depends on whether we consider the expression at the left of the `+` operator (say) to be directly the operand of the `+` or whether the conversion first applies, thence it is the resulting pointer which is "seen" by the `+` operator. The conversion does not take place irrespective of the operator the array *is an operand of*; hence, the array is an operand of the operator in question. Note that the semantics does make explicit that *if both operands have arithmetic type, the usual arithmetic conversions are performed on them,* hinting at the need of an explicit mention of arrays as a possible operands. The admitted reading is that it is not needed.

Whatever interpretation one chooses, with the introduction of arrays with selection, which are not converted to pointers, the text at the operator subclause must be explicit.

This also prompted the redefinition of the `[`*k*`]` array subscripting operator, which is no longer defined as equivalent to `(*((E1)+(E2)))`. This redefinition was the subject of another proposal already approved.

Non-lvalues cannot decay

The decaying of an array to a pointer to its first element is intrinsically impossible, in the abstract machine, if the array is not an lvalue. Some compilers allow it, giving rise, usually, to nonsense code. In the first place, it is difficult to create that situation if not doing it on purpose. Now there is are places where arrays may be operated, their value, and a non-lvalue array makes perfect sense there. It is as difficult as before to achieve this if not on purpose. The standard way (the only one?) is to call a function that returns a structure containing the array.

If functions are allowed to return arrays, avoiding the encapsulation-in-structure trick, the use of non-lvalue arrays becomes likely:

```
static const int base[3]={1, 2, 3};
typedef int int3[3];
int3 func(void);

func() == base[];
```

Here the comparison is testing whether `func` returned the `base` array.

Having to write `func()[]` to prevent the matrix from decaying to a pointer is a superfluous complication, because a value can only be used for its value and its type, not for its address because is does not have one. Therefore, we have changed the paragraph defining when an array is converted to a pointer, replacing *expression* by *lvalue*. In implementations defining meaningfully values as temporary objects with an address, the address of the array, or of its first element, can be obtained using the `&` indirection operator.

Literal 0 promoted to pointer

In a conditional operation, a common type has to be defined for the second and third operands. For this to be possible the types of those operands need be compatible or some rules have to be defined in case they are not. The latter happens for null pointer constants,

pointers in general and pointer to **void**. All combinations are possible except an ICE with the value 0 and the value **nullptr**. For arrays, we will require that the singletons of one and the other operand could be the operands themselves and define the common type accordingly. We exclude the combinations that need the value of the operand, not just its type; that is, we exclude singletons that are ICE with value 0 as possible match to a pointer or other null pointer constant (except, of course, if they match as integers, not as null pointer constants). The constant **(void*)0** need not be excluded because in any combination where it is allowed as null pointer constant (i.e., taking into account its value, not just its type) it is also allowed for its **void*** type.

The following are not allowed, amongst others:

```
constexpr int A[3] = {0,0,0};
constexpr void *B[3] = {NULL, NULL, NULL};
1 ? A[:] : B[:];
1 ? (int[3]){0,0,0}[:] : (void*[3]){NULL, NULL, NULL}[:];
```

The reason for not allowing this is that the reason for allowing them for scalar values is missing. They need be allowed for scalars because `0` is a common way of indicating a null pointer, so that, e.g., the following should be allowed:

```
p == q ? p : 0;
p == q ? p : NULL;
```

In the second line `NULL` might have been defined as `0`. But in the example above `A` is declared as an array of integers, so the values `0` it contains cannot be pointers. Here is an example of an allowed combination. Both arrays have elements of pointer type. `NULL` might have been defined as `0`.

```
float *A[3], *C[3];
A[:] = (float *[3]){&a, &b, &c};
C[:] = x > 0 ? A[:] : (float *[3]){NULL, NULL, NULL}[:];
```

We also exclude those combinations for the assignment operator in case the right operand is an array. Thus, the first assignment below is valid but the second one is not:

```
void *A[3];
A[:] = 0;
A[:] = (int[3]){0,0,0}[:];
```

A similar criterion is followed for the equality operators: if both operands are arrays with selection the restriction is the same as for the conditional operator. If one is an array with selection and the other is a singleton the latter may be a null pointer constant of integer type in case the array's singletons have pointer type or type **nullptr_t,** but the opposite is not allowed: if the singleton operand has pointer type or type **nullptr_t,** so must have the singletons from the array. The latter is also required in assignments but needs no extra wording because the combination of a left operand of integer type and right operand of pointer type or type **nullptr_t** is already not allowed.

Mixing arrays with selection and arrays which decay to pointers

Contrary to ANFV, we allow operating a matrix with selection with a matrix that decays to a pointer:

```
int *A[3], B[3], C[4][8];
A[:] - B;       // {A[0]-B, A[1]-B, A[2]-B}, with B converted to &B[0]
A[:] = C[1];    // A[i]= &C[1]
```

This affects the operators `+`, `-`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `=`, `+=` and `-=`. If the programmer wants this, the code can also be written avoiding the decay:

```
&B[0] + B[:];     or    (int*)B + B[:]
A[:] - &B[0];     or    A[:] - (int*)B
A[:] = &B[0];     or    A[:] = (int*)B
A[:] < &B[0];     or    A[:] < (int*)B
```

When writing `B[:]` (or `B[0:L:0]` for some `L`) instead of `B` is possible (with different semantics), allowing a `B` that decays to a pointer can make the code confusing. This situation arises when both a type *T* and a type "pointer to *T*" are possible as the operand. This can be the case for the operators of addition, equality and simpe assignment, though it is difficult to come up with a natural example; see the examples of this in the wording. The workaround, as shown in the examples above, is straightforward. So, it could be forbidden. However, conversion of arrays to a pointer to its first element takes place in C depending only on the operator, not on the other operand, and we prefer to keep this consistent. We recommended compilers to issue a warning in these cases.

On modifiable lvalues

The definition of modifiable lvalue excludes arrays from it. If we look for "modifiable lvalue" in the standard we find the following instances, aside from the definition:

- postfix/prefix increment and decrement
  [...] arithmetic or pointer type, and shall be a modifiable lvalue.
- *An assignment operator shall have a modifiable lvalue as its left operand.*
- errno
  which expands to a modifiable lvalue that has type int
- Checked integer operation type-generic macros
  result shall be a modifiable lvalue of any integer type other than ...
- stderr, stdin, and stdout
  are not required to be modifiable lvalues

Therefore, the only use of the term that needs arrays excluded from modifiable lvalues is the one on the assignment operator. But that subclause lists the possible combinations for the types of left and right operands, and an array can never be the left operand. So,

The exclusion of array from modifiable lvalues is not needed in the current standard

Hence, it would be more appropriate to say that an lvalue of array type cannot be modified because there is no production in the language that will do it, than to say that they are not, intrinsically, modifiable.

Now we have arrays that can be the left operand of an assignment, because it is possible that an array which does not decay is the right one. The obvious adjustment to the wording was to exclude from modifiable lvalues only arrays not carrying a selection. But

the preceding analysis shows that it is simpler to just drop the clause “does not have array type” from the definition. A further change is needed: it has to be required, for an array to be modifiable, that its singletons be modifiable lvalues.

This choice also accommodates arrays at the left of an assignment better, as in `E = B[]`, where `E` need not carry a selection.

9 Functions

Functions acting as range operators

Suppose we want to compute the square root of all the elements on the main diagonal of a matrix. The obvious way is to write

```
sqrt(N[0:n:n+1])
```

For this to work we need only treat a function call as any range operation. There may be more than one argument with or without range selection:

```
atan2(Y[0:n],X[0:n])
atan2(A[1][0:n],2.5)
```

The result is an array with selection, unless the return type is void; that is, the expression has array type and carries a selection. The selection is of the form of that of the arguments carrying the deepest selection; all subselections of the same depth must have the same length (number of elements), as in any range operation. The type of the singletons of the result is that of the return value of the function. This array can be used anywhere an array with selection can:

```
Y[:] = log(tan(0.5*F[:]))
Σ[:] = sqrt(N[0:n:n+1])
p[0:n][:] = exp(Ω[0:n][:])
```

The ordering of the sequence of function evaluations is unspecified, but the outcome shall match *some* order. E.g., in

```
void inc(float *p, const float x){
    *p += x;
}
float *p, A[6];
inc(p,A[:]);
```

the value in `*p` shall be incremented by the sum of all the elements in `A`. The translator is not allowed to parallelize the calls if it cannot ensure that.

This condition may be relaxed to, e.g., allow interleaved output. Implementation experience and user demands will tell.

If the innermost selected elements of some argument are matrices, these will be the values passed to the function. In this respect function calls behave like the operator `==`. A constraint will be broken because functions cannot take arrays as arguments, but this has nothing to do with the translation of a range call to individual calls.

If any of the function calls does not return, the behavior is undefined.

10 Selections of length zero

Motivation

If we want to switch the sign of the odd positions of a vector of size `n` we would write

```
A[1;n/2;2] = -A[1;n/2;2];
```

If `n` is 1 this results in a selection of length zero.

This breaks the rule that an array must have a positive length; i.e., must be an actual array. Should this be allowed or not? The answer follows from the observation already pointed above that the equivalent to a range instruction where the length of the selection is variable is not a VLA, but a "for" loop:

```
for(int i=1; i<n; i+=2) A[i]=-A[i];
```

"For" loops where the controlling variable is set to an initial value that already falls outside of the range of values to handle, and therefore execute no interaction, abound. The previous example is a typical one. Nobody would consider in the design of a language to preclude those loops from the language.

It would be easier, in the implementation of vectorization notation as proposed in this paper, to forbid zero-length selections. It would also be easier for the elaboration of the wording: we just mandate that the length be ≥1, as everywhere in C for arrays. These two arguments against zero lengths are a selfish, or lazy attitude on the part of the designers. That is not the way to design a language and, we would say, to design any product whatsoever. A programming language has to be designed from the point of view of the programmers, and from this point of view there is no possible justification to a programmer why he cannot write `A[1;n/2;2] = -A[1;n/2;2]` and must instead write

```
if(n/2>0){
    A[1;n/2;2] = -A[1;n/2;2]
}
```

It may be argued that a similar reasoning would lead to allowing zero-length variable length arrays, and they are not allowed and this is not widely felt as a missing feature. But VLA of length zero are analogous to memory reservations of size zero, and these are much less common than "for" loops that execute no interaction. Further, VLA are not widely used (many compilers do not support it) and, most notably among compliers that support VLA, gcc and Clang allow zero-length arrays. This may be why blessing of zero-length VLA by the standard has not been asked for more strongly, and not because users of VLA don't feel that they need it.

Relation to a value of B out of bounds

Allowing a value of 0 for `L` seems to imply that a value outside of bounds for the base `B` should also be allowed, even if given by an integer constant expressions, as is precisely the case in the example above. This in turn implies removing the constraint that the value for `B` shall fall within bounds when given by an ICE. This may seem dangerous, for a selection like

```
A[2];  A[2:L:s]
```

were `L` is not an ICE can only be valid if `L` evaluates to zero (if `L` is an ICE, the translator can detect it during translation). But this is already the case for "for" loops, and nobody thinks this is particularly problematic. Therefore, removing the constraint for range selections is not dangerous in practice either (or not more that many other situations that may result in out-of-bounds accesses).

But a third thought reveals that the situation analysed above is not subject to the constrain: The said constraint arises when the ICE `B` evaluates to a value *b* out of bounds *and* the length of the array is fixed constant. Situations with a variable `L` evaluating to a value of zero typically arise when the array has length *n* (variable) and the number of elements to process is a monotone increasing function of *n* that is zero for small values of *n*. Furthermore, in most cases we will have a pointer in place of `A`, not an array.

Therefore, in the end we keep the constraint. Code like

```
float A[1];
/* ... */
A[1;n/2;2] = -A[1;n/2;2];
```

that is subject to the constraint, may result from macro expansion. Fixing it is be very easy, and having it diagnosed causes more good than harm.

Its type

The obvious choice for it is an array of the element type, of length zero. Arrays of length zero do not exist in the language. Another option is to say that the expression evaluates to a void expression, but this proves untenable in the long term. For example, and supposing `L` evaluates to zero, the expression

```
sizeof(A[0:L][0])
```

is well defined and is the same as `sizeof(A[0])`, or as `sizeof(A[2])` in

```
float A[1];
sizeof(A[2]);
```

For another example, `A[0:L]==B[0:L]` evaluates to one (the condition is vacuously satisfied). All this is impossible if the expressions `A[0:L]` and `B[0:L]` have type **`void`**.

Fitting it into the type system

Contrary to broken arrays, an array of size zero is not something that can be disguised behind a qualification. Further, it does not disappear after lvalue conversion. Therefore, these range selections will make C have zero-length arrays. We can make so that they cannot escape the expressions where they arise, as we have done for broken arrays.

In the first place, let us review the restrictions present in the standard with respect to an array length equal to zero:

> **6.2.5 Types:** "An *array type* describes a contiguously allocated nonempty set of objects ... "
>
> **6.7.7.3 Array declarators:**
>
> > Fixed length arrays. **Constraint:** "If the expression is a constant expression, it shall have a value greater than zero."

Not fixed length arrays. **Semantics:** “otherwise, each time it is evaluated it shall have a value greater than zero.”

6.7.8 Type names: “... a *type name,* which is syntactically a declaration for a function or an object of that type that omits the identifier.”

Even though it is not mentioned explicitly, it is understood that all constraints and other conditions imposed on declarations also apply to type names.

The “nonempty” restriction on array types has to be removed. The restrictions in array declarators remain in place, so that arrays of length zero cannot result from a declaration.

There remains **typeof_unqual** and **auto** (arrays with range selection are precluded from **typeof**). For the first one, it would be enough to forbid arrays of length zero as its argument. The second requires a restriction on the type of the declared identifier:

- No identifier shall be declared of an array type with zero elements, or a type derived from it.
- No type name shall specify an array type with zero elements, or a type derived from it.

If this restriction is present, there is no need of another one to forbid arrays of length zero in typeof. However, we will not formulate this restriction as presented here, because it would overlap with the restriction for the value of array lengths. Instead, we will place one under **auto** and another one in range selections.

The restriction on type names could be relaxed to allow such a type name when this type, or a type derived from it, is the operand of a unary operator, thus allowing expressions like

```
sizeof(typeof_unqual(A[0:n-n]))
sizeof(typeof(typeof_unqual(A[0:n-n]) *))
(typeof_unqual(A[0:n-n])[3])
alignof(typeof_unqual(A[0:n-n])(*)[3])
```

But we feel there is no need to complicate the wording for these constructs’ sake. Range selection are not for placing them inside typeof.

Not allowing the above has the curious consequence that

`sizeof(A[0:n-n])`	is allowed but
`sizeof(typeof_unqual(A[0:n-n]))`	is not

In practice this is completely irrelevant: programmers will not write range selection for placing them inside typeof. Further, implementations are free to allow it. But the wording could also restrict the first form; that is, for operators that accept a type name and an expression, require that the type of the expression not be an array of length zero. This is what we will do in the end. In the first place, any code of the form `sizeof(A[0:L])` is written better as `sizeof(float[L])`, say. If the programmer insists in having a possibly zero `L`, the solution is to write `sizeof(A[0])*(L)`.

Literal zero for the length

We will forbid an integer constant expression of value zero (typically, a literal `0`) in place of `L`. This is not akin to forbidding an ICE `B` out of bounds. The latter will result in an out-of-bounds access unless *l* is zero, while an *l* equal to zero implies the opposite: there will

never be an out-of-bounds access (because no element will be accessed). We note, however, that an out-of-bounds access will take place if an array subscripting `[]` is applied to the range expression.

Many a programmer would prefer to permit an ICE with value 0 in place of `L`, for coherence: if an expression in place of `L` may evaluate to zero, why cannot if that expression is an ICE? They may be right, but coherence would imply allowing zero-length arrays essentially everywhere. It is not the purpose of this proposal to spread zero-length arrays in C. By forbidding them everywhere except in range operations with a non-ICE length they are a mild intrusion in the language. Further, implementations are always free to allow them, defining the obvious behaviour for those undefined behaviours. From a more practical point of view, a 0 in place of `L` can be protected with an `L ? ...` (where macro expansion will place a 0 in place of `L`), or can be forced by `(int)(0.0+(L))`, say.

Final set of restrictions

We will restrict arrays of size zero in places where type names are also possible, as noted above. This way we achieve a more coherent specification of allowed and not allowed operands. If we write *size* zero instead of *length* zero, there is no need to add "or a type derived from it" because an array with range selection cannot have the indirection operator applied to it; application of array subscripting `[]` creates either again an array or a singleton, and functions returning a pointer to an array of size zero (or further derivations) are impossible to express. Further, generic selections ignore array sizes if they are not integer constant expressions (and we are not allowing an ICE zero), so these arrays need not be precluded from those places.

Therefore, the final set of restrictions is:

- The initializer for an **`auto`** declaration shall not be an array of size zero.
- (in **range selections**) An array of size zero shall not be the operand of a **`sizeof`**, **`_Countof`** or typeof operator that is evaluated.

An array of length zero cannot be an operand of an evaluated array subscripting, but this follows from the fact that the subscript has to be within bounds.

We insist that implementations are free to allow zero-size arrays as operands to **`sizeof`** and **`_Countof`**.

The restriction to **`_Countof`** could be relaxed to *length zero* instead of *size zero*. But, even though the standard mandates that array lengths that do not change the value that **`_Countof`** "returns" not be evaluated, compilers do evaluate those array lengths. Forbidding size zero arrays aligns what is allowed for `E` in **`_Countof(E)`** with what is allowed in **`_Countof(typeof_unqual(E))`**. Finally, we note again that implementations are free to allow it.

Negative value for L

A value of zero for *l* results in an empty instruction. Likewise would a negative value. However, while performing zero interactions is well defined, performing -2 iterations, say, is nonsense. We won't allow them. If a user has an expression for `L` that may evaluate to a negative value, it can always be protected thus:

```
p[1:max(0,n-4)]
```

for some generic macro `max`.

Further, a negative value for L will conflict with a natural extension. See “Further extensions”.

11 IMPLEMENTATION OF RANGE OPERATIONS

What would be mandatory

As we noted in the introduction, small implementations are reluctant to adopt complex features, but at the same time the committee should strive to avoid divergence between implementations, which requires standardising common extensions and, as the present case, providing a proposal (technical specification) for an extension not yet common but for which interest has been expressed, and eventually make it standard, enriched with implementation feedback. The only way to achieve both goals is to standardise the features and at the same time allowing implementations not to implement them if they don’t want.

In the first place, we do not want to make range selections mandatory:

`__STDC_RANGE_SELECTIONS__` Undefined or defined and expands to `0` if range selections are not supported; expands to `1` otherwise.

Note that we have written *range* selections. We intend to make empty selections mandatory.

The established practice for feature test macros is to define the macro in negative terms:

`__STDC_N0_RANGE_SELECTIONS__` Defined to `0` if range selections are not supported.

This way, if in the future the feature is made mandatory, the macro can disappear. Actually, this can also be done if the macro is defined in positive terms. If so, programs will need a test for the version under which they are being compiled if they want to test the availability of the feature, since lack of the macro can mean very old or very new version. But a definition in negative terms also requires a test for older versions. In one case the version test is to distinguish before mandatory from after mandatory; in the other case, before existence from existence.

There may be a difference in the expression of intent; a negative definition seems to convey that supporting the feature is the default. We choose the positive definition because we will be using the macro also for another purpose, where absence of the macro cannot possibly mean the new behaviour (see “Conflict with array subscripting” below).

Implementations defining `__STDC_ARRAY_SELECTIONS__` to `1` may still be allowed to only accept a partial set of possible combinations. We propose the following optionality:

`__STDC_ARRSEL_NESTED__` Expands to `0` if selections of the form `[B:L]` and `[B:L:s]` can only be applied to expressions of pointer type and to arrays carrying no selection or an empty selection, and range operations are allowed only if the selected elements from the matrices treated as a range of objects are singletons. Expands to `1` otherwise.

`__STDC_ARRSEL_STEPPED__` Expands to `0` if stepped selections are not supported; to `1` if they are supported.

If these two macros expand to `0` broken selections cannot arise. Furthermore, a value of `0` in the first macro means that a maximal sequence, in the code, of adjacent array selec-

tions either results in a selection of singletons or is an empty selection (i.e., `[][]... []`), except that a partial selection is allowed if it is used as the operand of some operator that treats it as a whole, namely, **`sizeof`** or the typeof operands. Such an implementation would allow comparisons of whole matrices as in `A[] == B[]` and assignments like `E = B[]`, but not assignments like `E[:] = B[]`, which is a range operation on the elements `E[:]`, which are not singletons.

A value of `0` in the two macros constitutes a considerable simplification for the implementation with respect to the full set and yet it offers the programmer the most common use cases of range selections. It becomes even more useful if combined to casts that redimension the matrix:

```
int A[4][4][20];
((int[16][20])A[])[0:8];
```

thought this is of secondary importance.

More macros or more values for the proposed macros could be defined, but a fine -grained possible support expressed via macros would be of no use if the programmer knows what his implementation supports, or a burden for the programmer if he wants to produce a strictly conforming program. In the latter case he may cut short and just test for support in a specific form ignoring any other combination.

As regards a step zero, we gradually shifted our preferences from including it in `__STDC_ARRSEL_STEPPED__`, which could then take values `0`, `1` or `2`, to require the support for step zero whenever stepped selections are supported. This happened because of our thinking of how an implementation may translate range operations.

Another reason for not allowing more complicated yet partial implementations of the feature is that, if a translator finds a certain selection too complicated it can always translate it into a “for” loop. Thus, either provide a simple set of features or provide the full set.

In addition, we have thought of another macro expressing partial support:

`__STDC_ARRSEL_CONSTANT__` Expands to `0` if `L` and `s` need be integer constant expressions as well as `B` if the selection applies to an array already carrying a selection. Expands to `1` otherwise.

This macro `__STDC_ARRSEL_CONSTANT__` becomes more significant when *direct selections* are included. If this happens and the macro is included, the values might be reversed with respect to the proposal above, to make `0` mean the same as its absence.

How range selections might be translated

Take as example the following instructions:

```
float A[6];
A[:] = B[0:6] * C[0:6:2];
A[0:4] = B[4:4] * s;
```

When the compiler sees the selection `[:]` from `A` it has parsed a selection of 6 elements. Therefore, any arrays following in the instruction (and not broken by `,`, `?` `&&`, `||` or within unary, not arithmetic operators), if they carry a nonempty selection they must carry a selection of six elements. The compiler may parse all selections, check that they are of six elements if given by ICE, ignore those that are not ICE (assuming its value is 6) except for their side effects, and prepare to translate the instruction into vectorial processor instructions or a “for” loop. In this case,

```
for i=1..6  A[i]=B[i]*C[2*i]
```

In the second statement, in addition to operands which are arrays carrying a selection of a matching number of elements, four in this case, one of the operands is a singleton:

```
for i=1..4  A[i]=B[i]*s
```

The same scheme can be applied when the selections have different depth:

```
float A[6][9], C[6];
A[:][:] *= C[:];
```

Which gets translated to

```
for i=1..6  A[i][:]*=C[i]
```

which in turn is translated to

```
for i=1..6
    x=A[i]. y=C[i]
    for j=1..9  x[j]*=y
```

The instruction may also include an array with an empty selection. This is treated like a singleton, and when it appears in the “for” loop the operands it is operated with must be matrices of the same dimensions and the operation is performed elementwise:

```
float A[6][9], C[9];
A[:] *= C[];
for j=1..6  A[i]*=C[]          -->          for i=1..6
                                                x=A[i]
                                                for j=1..9  x[j]*=C[j]
```

except that `C` is evaluated only once.

```
float A[6][9][3], C[9][3];
A[:] *= C[];
for j=1..6  A[i]*=C[]          -->           for i=1..6
                                                 x=A[i]
                                                 for j=1..27  x[0][j]*=C[0][j]
```

The previous analysis is a very cursory one. We can nonetheless already draw an important conclusion from it: the selections and different operands that constitute an instruction may get evaluated in many different orderings. In particular, the option to first evaluate all range selections is one obvious one.

Sequence points

The chain of operands that must carry selections of the same length (if not empty) is broken by the operators

```
,   ?   ||   &&
```

in addition to operators that enclose their argument (**`sizeof`**, etc., `[]` subscripting and

`_Generic`). Those four above are, not by accident, the ones that introduce sequence points. The idea put forward elsewhere for fixing the order of evaluation in operations like `A+B` has to consider carefully its implications for range operations. This may invalidate the strategy of first evaluating all range selectors.

It would also break the commutativity of operators that are currently commutative. For example, `i++ * i` would yield a different value than `i * i++`. An alternative that fixes the order of evaluation of `++` and `--` with respect to the operands and at the same time preserves commutativity, consists in performing all postfix inc./dec. after the operation and all prefix inc./dec. before. An example where this matters for range operations is

```
p[0:n]++ * p[0:n];    p[0:n] * p[0:n]++;
```

These two instructions would have defined behaviour and yield the same value.

The previous example is not specific to range operations, to the problem pointed above that evaluation of all range selectors first would not be possible if `*` introduced a sequence point and the left operand had to be evaluated before the right one. An example that does exhibit this problem is

```
p[0:p[0]] * p[0:n]++;    p[0:n]++ * p[0:p[0]];
```

Mandating here the evaluation of the left operand before the right one would preclude the translation of range selections in the first place in the instruction at the right. It would also imply that the first instruction is well-defined if `p[0]` equals `n` before the instruction, while the second one is well-defined if `p[0]` equals `n-1`. Moving the increments to after the multiplication gives both expressions the same, well-defined, semantics. Analogously,

```
p[0:p[0]] * ++p[0:n];    ++p[0:n] * p[0:p[0]];
```

have both a well-defined, equal, semantics, different from the previous ones.

A problem of a different kind is exemplified by the following instructions:

```
p[0:++n]*q[0:++n];    p[0:n++]*q[0:n++];
```

Here, moving the `++` increments to before the product, in the first case, and to after it in the second case, implies moving them through the range selector (the `[]` brackets). We think that a better choice is that these brackets, whether used for a range selection or for array subscripting, introduce sequence points. Evaluation of the expressions within the brackets would precede that of the selected elements from the matrix (currently, the value computation is already mandated to take place before, but side effects are not) but would be subsequent to the evaluation of the array expression (the postfix expression to the left of `[]`). This would make defined expressions like

```
A[A[0]++];    p[0:p[0]++]*=2;
```

The product instructions above would still remain undefined (the value read for l, the range selection length, may be anything), but we think it is better to preserve commutativity, even if it means keeping some cases undefined.

For assignment operators the situation is similar:

```
A[0:n-1] = B[0:*p++];
```

Here, if the `=` sign introduces a sequence point (as would be the case for a `*` operator in place of `=`), the compiler is not allowed to read first `n-1` and `*p` (or rather, to compute `n-1` and ignore `*p`); it would be forced to update first `*p`, since `p` could point to `n`.

The problem is not limited to artificial examples. “Normal” instructions like

```
A[n:5] = B[*p++:5];
pa[i:n] = pb[*q++:n]*pc[i:n];
```

cannot be parallelized easily if `*` and `=` introduce sequence points, because (taking the second example) `q` could point to `i`, or within the ranges `pc[i:n]` or `pa[i:n]`.

The intrusion into parallelization of the sequence point does not need the evaluation of the expressions in the selectors:

```
A[0:n] = B[0:n]*C[0:n];
```

If the evaluation of `B` or `C` had side effects on `A`, the compiler would be forced to store the whole `n`-element array in a temporary place. This is of course not the case in this example, but it is easy to replace `B` or `C` by something that could have side effects on `A`. We could make `B` or `C` equal to `A` itself and make it volatile, forcing the compiler to really read all `B`, say, before starting storing in `A`. Or we can replace `B` by a range function call or a function returning an array (or, as long as the latter is not possible, returning an array encapsulated in a structure):

```
A[0:n] = func(B[0:n])*C[0:n];
A[0:6] = func(x)*C[0:6];   //Suppose func returns an array of length 6.
```

There are many ways that `func` could modify `A`: `B` can hold pointers into `A`, etc.

If forcing a strict order of evaluation of the range selectors and the matrix themselves can be a hurdle for the translator, having to store an intermediate array, possibly of variable size and possibly large, is out of the question. We noted that variable length range selections are analogous to “for” loops, not to memory allocations as is the case for VLA. Making `=` and other operators into hard sequence points plays havoc on this.

The root of the problem is that parallelizing is the opposite to sequencing (harmony vs. melody). The solution may be to make assignment operators introduce sequence points *except* when the assignment operation is a range operation. If the assignment is of the type *array with range selection = singleton,* the problem does not arise, so this case can be excepted (from the exception itself). Or to avoid those exceptions, that the operator is a sequence point except when the right operand is an array that does not decay to a pointer.

But there is a better solution. First, sequence points should not interfere at all with parallelization. Second, they can still apply to each individual singleton assignment. Thus, in the example above

```
A[0:n] = func(B[0:n])*C[0:n];
```

in each singleton assignment `A[`i`] = func(B[`i`])*C[`i`]` the operator introduces a sequence point, and the evaluation of the right operands is sequenced before the evaluation of the left one. The evaluation of the different singleton assignments shall remain unsequenced (though in this example including a range function call, the different calls to `func` are indeterminately sequence). This implies that if some call of `func` modifies some element of `A` other than its corresponding one the behaviour is undefined. This is intrinsic to parallelization. Trying to avoid this destroys the performance benefits of these range operations. Still, side effect become well defined:

```
p[0:n] = p[0:n]++
```

Here, each `p[`i`] = p[`i`]++` is well defined because of the sequence point and does not inter-

fere with the other assignments.

An exception is needed if the possibility of *s. += arr.* is added:

```
s += a[0:n]*b[0:n]
```

Here, because the left operand, a singleton, is evaluated only once, it is obviously impossible that it is evaluated after each of the singletons at the right, without making the **+=** the hard sequence point to be avoided. Since, as we saw, this kind of compound assignment should have sequencing properties like those of range function calls, the evaluation of its operands and the individual singleton assignments should be sequenced like that of the arguments and actual calls of a range function call, as though the left and right operands were function arguments.

Finally, making : introduce a sequence point and mandating left to right evaluation would probably pose no serious problem, thought we would prefer to wait for implementation feedback. It would interfere with the strategy of evaluating all range lengths in sequence, not evaluating anything in between. For example, if a range operation contains the range selections `[B:L]`, `[B':L']` in that order, and maybe others following, and `L'` is an integer constant expression, evaluation of `B'` might modify the value computed for `L'`, and the compiler is not allowed to evaluate `L'` right after `L`. (If `L'` is not an ICE nor has side effects, the translator can in any case omit its evaluation, for its value should be equal to that of `L`).

12 Indexed and direct selections

These are not proposed to be included in the specification. They may be provided by implementations as an extension and be eventually incorporated. See the document ANFV for their analysis. We only note here the different kinds of selections proposed.

The different kinds of array selections

Terminology

Empty selector: `[]`

Range selector: `[B:L]`, `[b...e]`, `[B:L:s]`, `[:]`, `[::]`

Indexed selector: `[I]` (`I` may or may not carry a selection)

Direct selector: $[K', K'', \ldots, K^{(l)}]$, $[\{k^{(1)}, \ldots, k^{(m)}\}', \{k^{(1)}, \ldots, k^{(m)}\}'', \ldots]$

Here `B`, `L`, `s`, `k` stand for expressions of integer type, `b`, `e` for integer constant expressions (hence, `b...e` is a constant range expression, CRE), `I` for an expression of array type one- or two- dimensional and `K` for either an expression of integer type or a CRE.

Note that we have included a single CRE in range selectors.

The most general selection carried

As a result of the combination of all possible range selections, an array carrying a non-empty selection will always carry a certain selection in each dimension, for a number *m* of dimensions starting from the outermost one, which we have called the depth. In each dimension the kind of selection is one of *****, $\emptyset$, $b{:}l$, $b{:}l{:}s$ or *irregular*, where ***** means "all elements are selected", $\emptyset$ means an empty (in length, nor in depth) selection and an irregular

selection is given by the index array or direct selection used for it, in which case the length of the dimension of the array with selection it applies to is the product of the lengths of the dimensions involved in the original array (see ANFV for more details).

Comma-separated list

A selection like `A[0,2,n]` selects elements in the obvious way:

```
float A[4][4];
A[0,2,n];            // {A[0], A[2], A[n]}
```

Side effects should not be allowed in the first place. If they prove useful, the restriction can always be lifted; the reverse would be very difficult, and there are plenty of places in C where we would like the language not to be as permissive as it is, but now it is too late.

Conflict with array subscripting

We have designed (in ANFV) the syntax of comma-separated lists as selectors as we would like it to be. But there is an obvious problem: the expression in a subscript selection may include the comma operator. That is, currently `A[3, 0, 1, 5]` is interpreted as `A[5]`. There are several possible solution to this.

1. To add a pair of `{}` enclosing the list. This is the most obvious and straightforward solution. However, this would conflict with uses of `{}` for >1-depth selections; e.g., `A[{1,2},{2,3},{3,3}]`. Here, three elements are selected. If we select only the first one, it will look like `A[{1,2}]`. To avoid these, these selections would need an extra `{}` too: `A[{{1,2},{2,3},{3,3}}]` and `A[{{1,2}}]`. See ANFV for more details.

   Other than being cumbersome, this also goes against common practice everywhere that a comma-separated list enclosed in some brackets represents a list of selected elements (the list is already enclosed by the `[]` brackets).

2. To let `,` inside the `[]` operator be interpreted like the comma operator or like the comma of a direct selection list according to some pragma or other directive specifying the version of the language, as has been proposed. For example

   ```
   #STDC_features 35
   ```

   If the value is < 35 the comma is interpreted as the comma operator and direct selections are not allowed; otherwise, the expression k in `[`k`]` for the subscripting operator is a conditional expression.

3. A directive specific for this situation. For example,

   ```
   #STDC_features ARRAY_COMMA 0
   ```

4. Use the `__STDC_RANGE_SELECTIONS__` macro. If it is defined to `1`, the comma stands for list separator.

5. Let it be implementation defined but not tied to the `__STDC_RANGE_SELECTIONS__` macro.

Since an implementation supporting direct selections must also support stepped selections, we could have taken the value of the macro `__STDC_ARRSEL_STEPPED__` in solution 4 as the watershed. But the purpose is not to push comma operators inside array subscripting as far as possible when using array selections, but rather the opposite. That if

somebody wants to use array selectors he cannot place a comma operator inside an array subscript seems a mild requisite and even desirable.

A comma operator only makes sense if the first operand has side effects, which are not allowed in direct array selectors. Hence, in solution 4, compilation with `__STDC_RANGE_SELECTIONS__` defined to `1` will make those expressions trigger an error (and similarly for solutions 2, 3 and 5). The safest path towards the elimination of those expressions is as follows: first, the code is compiled with a new compiler, with array selections turned off (whereby a comma stands for the comma operator), and asking the compiler to issue a warning for array subscripts that are comma expressions (a warning likely to be provided by compiler supporting array selections, when these are turned off). Those expression are removed or parenthesised, e.g. `[(p++,n)]`. Then code is compiled with array selections on; i.e., with `__STDC_RANGE_SELECTIONS__` defined to `1` and commas added henceforth within `[]` are interpreted as list separators.

Furthermore, array subscripting including a comma operator is extremely unlikely to appear in header files; there may be none at all.

As to code written with array selections and attempted to be compiled with an old compiler: 1st, new features are not supported by old compilers, this one is like any other. 2nd, a silent, wrong compilation can be avoided by warning on expressions with no side effects; in this case, the first operand of the comma operator, as the old compiler would interpret it.

13 Further extensions

A[B:-L]

One may expect implementations to allow a negative `L` in `A[B:L]`, write it `A[B:-L]`, as a synonym of `A[B:L:-1]`. Maybe restricted to ICE.

A[B:L][B':L'] when the elements of A[B:L] are pointers

As in

```
int *B[10];
B[2:4];          //Array of four pointers
B[2:4][0:3];     //[a,b,c]  [a',b',c']  [a'',b'',c'']  [a''',b''',c''']
```

Allowing constructions like the latter might add a substantial burden to implementers. It seems these would have few use cases. It may on the other hand seem useful. Actually, both perceptions are not contradictory. We think it is better to have implementations first implement multidimensional selections just on multidimensional arrays, then allow or not this further extension based on their feedback.

This restriction is also in force in OpenMP.

Functions returning arrays

Now that there is a way of preventing a matrix to decay to a pointer, an array may be returned: `return a[]`. The syntax for the declaration is already part of the language:

```
typedef double dbl33[3][3];
```

```
dbl33 VdM3(double a, double b, double c){
    double A[3][3], *p=&A[0][0];
    *p++=1; *p++=1; *p++=1;
    *p++=a; *p++=b; *p++=c;
    *p++=a*a; *p++=b*b; *p++=c*c;

    return A[];
}
```

The declaration can be written without the need of a **typeof** following the rule that a declaration mimics the use:

```
double VdM(double a, double b, double c) [3][3];
```

There is a constraint forbidding an array as the return type. It need only be removed for this extension to come into being. The declaration that does not make use of **typeof** needs a further modification of the wording for array declarators. The return type should still be restricted to be of fixed constant size: “A function declarator shall not specify a return type that is a function type or a variable length array type”.

Keeping the constraint seems unnatural; the more, since the return type may, currently, be a struct with an array as its single member. Hitherto this was virtually useless, because an array that is not an lvalue cannot be operated with anything other than the subscripting `[]` operator, or used only for its type (**sizeof**, etc.):

```
struct S{float a[3];};
struct S func(float);
float f = func(1.0f).a[0];
```

But now it can be assigned to, added to another matrix... Having to encapsulate the returned matrix in a structure is absurd:

```
p[0:3]= func(x).a;
func(x).a + func(y).a
```

etc.

Functions taking arrays

Likewise, arrays may be passed as arguments. There is still need to invent a syntax for parameters of array type, as in the following example:

```
double determinant3(double A[:3][3]);
/* ... */
double M[3][3];
determinant3(M[]);
```

The obvious choice for the parameter syntax and, we would say, the right choice, is `double A[3][3]`, which we all know is not possible.

Here is an example that both takes and returns a matrix:

```
typedef double dbl33[3][3];
```

```
dbl33 invert3(double A[:3][3]){
    /* Compute A^-1 and store it in A */
    return A[];
}

double M[3][3], N[3][3];
N= invert3(M[]);
```


6.5.16 Conditional operator

The paragraph on the determination of the common type when both operands are pointers or **nullptr_t** or a null pointer constant (p. 6 currently, p. 9 in our proposal) is too convoluted as a result, we presume, of incremental editing. We have simplified it. No semantic change is intended.

6.5.17 Assignment operators

The paragraph restricting the overlapping of the assignee with the read object had to be modified to make it apply only to singletons, and this required some minor adjustments to the sentence. We have modified it further so as no to speak of the type of an object, but of the expression used to access it. There is a proviso near the beginning of the standard to make "the type of an object" mean that of the identifier used to access it, but since here we are referring to an object referred to by two identifiers (the one at the left of the assignment and the one at the right), it seems that a direct mention of the type of the identifiers is clearer:

> If the left operand is not an array and the value being stored in it is read from another object that overlaps in any way its storage, then the two objects shall occupy exactly the same storage and the type of the expression used to access the object read shall be a qualified or unqualified version of a type compatible to that of the left operand; otherwise, the behavior is undefined.

That paragraph is placed under "simple assignments" (it is par. 3), while it applies to any kind of assignment. It is true that compound assignments are described as equivalent to a certain simple assignment, but we feel it would be clearer if the paragraph were "promoted" to assignment operators in general.

Our proposal adjoins one paragraph and several examples to that paragraph 3, and those should be moved together with it if it were moved. We have not done so.

WORDING

Here text is provided for empty and range selections. Indexed and direct selection are not considered. The terms *range operation* and *depth of selection* were finally not introduced because they did not seem necessary for the wording. The latter would simplify the wording in just one place: the phrasing of a constraint for assignment operators.

Blue text is new text, green one is changed text, ~~gray~~ text is to be removed. Margin notes are comments to the wording but not part of it. Within them, A[:] stands for an array with nonempty selection; A[:]s for an array carrying a selection of singletons; A[] for an array carrying an innermost selection of arrays, B for an array with no selection or empty selection, and s for a singleton.

6.2.5 Types

25 Any number of *derived types* can be constructed from the object and function types, as follows:

— An *array type* describes a ~~contiguously allocated nonempty~~ possibly empty[xx] set of objects with a particular member object type, called the *element type*. If the array is not broken the objects are allocated contiguously. The element type shall be complete whenever the array type is specified. Array types are characterized by their element type, ~~and~~ by the number of elements in the array and, if the array is broken, by the relative memory position of its elements. [...]

NOTE The semantics of a program never depend on whether the type of an array is broken or not. Thus, a translator may ignore this fact. In contexts where the qualified type of an expression ("broken" is a qualification) is needed (e.g. for **typeof**), an expression that can potentially designate a broken array is not allowed. For practical purposes for translators, the broken qualification could be omitted. Formally this is not possible: the type of broken arrays cannot be the same as that of ordinary arrays because they have a different object representation.

27 An object or value which is not of array type is called a *singleton*. If the element type of an array is not an array type, the elements of the array are *its singletons*. If the element type is an array type, the singletons of the array are those of its elements.

6.2.6 Representation of types

6.2.6.1 General

2 Except for bit-fields and arrays of size zero, objects are composed of ~~contiguous sequences of~~ one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined. If further the object is not a broken array, the bytes form a contiguous sequence.

3 [...]

4 ~~Values stored in non-bit-field objects of any other object type~~ Except for bit-fields and broken arrays, values stored in an object of a type are represented using $n \times$ **CHAR_BIT** bits, where n is the size of an object of that type, in bytes. An object that has the value can be copied into an object of type **unsigned char** [n] (e.g. by memcpy); the resulting set of

[xx] An empty array can only be created by a range selection of length zero. No identifier can be declared with this type or a type derived from it.

bytes is called the object representation of the value. Broken arrays differ from ordinary arrays in the relative position of its singletons, which is defined by the range selections that give rise to it (6.5.4.3), not in the way the singletons are represented. Their size is considered to be the same as that of the corresponding ordinary array, even if some singletons share the same memory location. Values stored in bit-fields [...]

6.3.3.1 Lvalues, arrays, and function designators

1 An *lvalue* is an expression (with and object type other than **`void`**) that potentially designates an object;[54] if an lvalue does not designate an object when it is evaluated, the behavior is undefined. When an object is said to have a particular type, the type is specified by the lvalue used to designate the object.

2 A *modifiable lvalue* is an lvalue that either:

— Does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type; or

— is an expression `E` with array type and `*(E)` is a modifiable lvalue (if this expression has array type the rule applies recursively).

3 Except when it is

— the operand of the **`sizeof`** or typeof operators,

— the operand of the the `++`, `--` or unary `&` operator, or

— the left operand of the `.` operator or an assignment operator,

an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called *lvalue conversion*. If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; additionally, if the lvalue has atomic type, the value has the non-atomic version of the type of the lvalue; otherwise, the value has the type of the lvalue.

4 An lvalue which is an array with selection undergoes lvalue conversion in the same contexts as singleton lvalues. The result is an array with the same selection and with its singletons having undergone lvalue conversion as described above.

"with its singletons" instead of "with its elements" because "described above" is only for singletons.

5 EXAMPLE

```
int A[6][4];
const int B[10][10];
A[:][:] = B[2:6][0:4];
```

When the expression `B[2:6][0:4]` undergoes lvalue conversion the result is an array of type **`int`**`[6][4]` where each singleton results from the lvalue conversion of the corresponding singleton in `B[2:6][0:4]`.

6 If an lvalue ~~(that does have array type)~~ designates an object of automatic storage duration that never had its address taken, and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use) when it undergoes lvalue conversion, the behavior is undefined. An address of an object is taken by application of the address operator to an lvalue designating the object, by being an element of an array that was designated by an expression that is converted to a pointer as described below, or by being a member of a structure or union object whose address is taken.

7 An ~~expression~~ lvalue that has type "array of *type*" and the array does not carry a selection

is converted to an expression with type "pointer to *type*" except when it is:

— the operand of the **sizeof**, **_Countof** or typeof operators,

— the operand of the unary **&** operator or the left operand of an assignment,

— one of the two expressions of an array subscripting operator or the left operand of a range selection, or

— a string literal used to initialize an array.

The resulting pointer points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is implementation-defined.

8 [...]

Forward references: address and indirection operators (6.5.5.3), assignment operators (6.5.17), common definitions **<stddef.h>** (7.21), initialization (6.7.11), array subscripting (6.5.4.2), array selection (6.5.4.3), postfix increment and decrement operators (6.5.4.6), prefix increment and decrement operators (6.5.5.2), the **sizeof**, **_Countof** and **alignof** operators (6.5.5.5), structure and union members (6.5.4.4).

6.5 Expressions

6.5.2 Arrays with selection as operands

Constraints

1 For the operands in the following two cases:

— the two operands of the multiplicative, additive, equality, relational and assignment operators, and those of all binary bitwise operators, when both of them are arrays carrying a nonempty selection;

— the arguments of a function call that are arrays carrying a nonempty selection, when these are more than one,

the lengths of the outermost selections, if both of them are fixed constant, shall be the same.[xx)]

2 For the operands in the first case with exception of those of equality operators, if one of the operands carries a selection of singletons (i.e., with selection in as many dimensions as the array has), so shall the other.

A[:]s == B[] is allowed

3 For the binary operators above with the exception of relational and equality operators,[xy)] if one operand is an array with an empty selection or without selection which is not converted to a pointer (e.g., as a result of the application of the rule in p. 5 below), the other operand shall be an array that is not converted to a pointer; if this array carries a selection, its innermost selected elements shall be arrays. When these conditions for the operands hold then, also for relational operators, those innermost arrays of the other operand or the operand itself if it does not carry a selection shall have the same number of dimensions as the first mentioned operand (considered both as multidimensional arrays), and if for any of these dimensions the lengths in the two operands are fixed constant, they shall be the same.

A *op* *x*. *x* must be B or B[], except that A == s is allowed. The dimensions of A and B or B[] shall match.

[xx)] The length of inner selections shall also be the same. This follows from the recursive rule in p. 5 below.

[xy)] For relational operators a stronger requirement holds: no operand can be of the kind described in this paragraph (6.5.10).

4 EXAMPLE 1 In the code

```
int A[10][5][n], B[5][9];
A[:] + B[];
```

the addition expression satisfies the constraint: One of its operands, namely `B[]`, is "an array with an empty selection" and the other operand, `A[:]`, is "an array that is not converted to a pointer". This second operand carries a selection and satisfies that "its innermost selected elements shall be arrays". Further, the innermost selected elements from this other operand; i.e., each of the $A[i]$, are arrays with the same number of dimensions (two) as "the first mentioned operand", `B[]`, and of these two dimensions the first one is fixed constant in both each $A[i]$ and `B[]`, and the lengths agree (5).

Semantics

A[:] *op* A[:]

5 If the operands of the binary operators listed in the first constraint are arrays carrying a nonempty selection, when their lengths are evaluated they shall be the same. The expression is an array of that length, with its elements selected, where each element is the result of operating the corresponding element from the first operand with the corresponding element from the second operand. For this operation, if the elements of any of the operands are arrays with no further selection they are treated as if they carried an empty selection (that is, they are not converted to pointers).

A[:]s *op* s

6 For the same operators as in the previous paragraph, if one operand is an array carrying a selection of singletons and the other operand is a singleton or an array which is converted to a pointer, in which case the latter pointer is a singleton, the expression is an array with the same dimensions as the operand which is an array with selection, with its elements selected, each element being the result of operating the corresponding element from the array with selection with the other operand, according to the semantics described for the operator for operands which are singletons. This other operand is evaluated only once.

A *op* B
A[] *op* B

A == B produces a singleton, not an array

A < B not allowed

7 For the same operators as above with the exception of relational and equality operators, if one operand is an array with an empty selection or without selection which is not converted to a pointer, let `E` be the other operand, and let v be the innermost selected elements from `E` or the operand `E` itself if it does not carry a selection. The result is an array with the same dimensions and selection as `E` and where the innermost selected elements are the result of operating each singleton of the corresponding v with the corresponding singleton of the other operand; each singleton of this latter operand is evaluated only once. Whenever the expression is evaluated, the array lengths of v and of the first mentioned operand (considered as multidimensional arrays) shall be the same.

8 The dimensions of the resulting array are given by the dimensions of the operated arrays, as expressed in the previous paragraphs. When at least one of the array lengths of the operands that correspond to an array length of the result is fixed constant, the corresponding array length of the result is fixed constant.

9 Whenever an array has each of its elements operated with the other operand, or with a corresponding element of the other operand, these operations are unsequenced.

10 Whenever an array has each of its elements operated, if none of the individual operations would raise a certain floating-point exception, that exception is not raised. If any of the operations would raise it, it is implementation defined whether the exception is raised or not. The corresponding status flag is set or not accordingly.

11 EXAMPLE 2

```
short A[12], B[12];
int C[8][12][4], D[8][10][2], b;
```

```
B[0:6] = A[0:6] + A[6:6];
D[:][0:6][:] = C[:][0:6][0:2] + C[:][6:6][0:2];
```

In the expression `A[0:6] + A[6:6]` each of the operands is an array of six elements carrying a selection. The expression is therefore an array of type `int[6]` with its elements selected and whose values are the sums of the two arrays. The type of the assignment expression of which it is part is also `int[6]`. In the expression `C[:][0:6][0:2] + C[:][6:6][0:2]`, each operand as well as the expression itself and the assignment expression of which it is part are of type `int[8][6][2]` and carry a three-dimensional selection. The operated elements are `C[`i`][`j`][`k`] + C[`i`][6+`j`][`k`]`. The 96 operations are unsequenced.

12 EXAMPLE 3

```
float A[n], B[n][n], C[n];
A[:] = B[:][:] * C[:];
B[:][1:n/2:2] = -B[:][1:n/2:2];
```

The rule in p. 5 applies to `B[:][:] * C[:]`, yielding `B[`i`][:] * C[`i`]`. Each of these is the multiplication of an array with singletons selected, `B[`i`]`, and a singleton, `C[`i`]`. Therefore, the rule of p. 6 applies to it; the result of each operation is the value `B[`i`][`j`]*C[`i`]`, with $0 \le j \le n$, where n is the value to which `n` evaluates at the declaration, and that of the multiplication expression is the $n \times n$ array `B[`i`][`j`]*C[`i`]` (multiplication of `B` by rows).

13 The expression `B[:][1:n/2:2] = -B[:][1:n/2:2]` is likewise subject to the rule in p. 5, yielding `B[`i`][1:`n`/2:2] = -B[`i`][1:`n`/2:2]`. For each of these, the rule in p. 5 applies again, resulting in `B[`i`][`j`] = -B[`i`][`j`]` for odd j. Here n may be 1, in which case no operation is performed and the expression has type `float[1][0]`.

14 EXAMPLE 4

```
float A[n][m][5], B[n][5], C[5], D[m][5];
B[:] *= C[];
A[:] *= D[];
A[:][:] *= B[:];
```

The rule in p. 7 applies to `B[:]*=C[]`, whereby each `B[`i`][`j`]` is multiplied by `C[`j`]` (multiplication of `B` by columns). By the same rule, each `A[`i`][`j`][`k`]` in the next statement is multiplied by `D[`j`][`k`]`. In the third statement, application of the rule of p. 5 yields that each `A[`i`]` is multiplied by `B[`i`]`; each of these multiplications is like that of `B[:]*=C[]` (with `m` in place of `n`).

15 EXAMPLE 5 The code

```
int A[10][10];
A[:] + 1;
```

has undefined behavior. While `A[0] + 1` through `A[9] + 1` would be valid, when these operations are part of a range operation as in `A[:] + 1`, each `A[`i`]` is not converted to a pointer and there is no semantics defined for the addition of an array and an integer.

16 EXAMPLE 6 The additive expression is an array of fixed constant length:

```
int A[6], B[n];
A[:] + B[:];
```

6.5.4 Postfix operators

6.5.4.1 General

Syntax

postfix-expression:

primary-expression
postfix-expression `[` *expression* `]`
postfix-expression `[ ]`
postfix-expression *range-selector*
postfix-expression `(` *argument-expression-list*$_{\text{opt}}$ `)`
postfix-expression `.` *identifier*
postfix-expression `->` *identifier*
postfix-expression `++`
postfix-expression `--`
compound-literal

argument-expression-list:

assignment-expression
assignment-expression-list `,` *assignment-expression*

range-selector:

`[ : ]`
`[` *conditional-expression* `:` *conditional-expression* `]`
`[` *conditional-expression* `:` *conditional-expression* `:` *conditional-expression* `]`

6.5.4.2 Array subscripting

7 EXAMPLE 2

```
int x[6];
x[2:3][0]; // Designates the element x[2]
```

6.5.4.3 Array selection

Description

1 An empty pair of brackets following a postfix expression is an *empty selector*. A postfix expression followed by an empty selector is an *empty selection*. Its intent is to select the matrix as a whole.

2 A postfix expression followed by a construction of the form `[:]` or `[B:L]` or `[B:L:s]`, where `B`, `L` and `s` are conditional expressions is a *range selection,* which selects some elements from the array. The intent of `[:]` is to select all elements, that of `[B:L]` to selects `L` elements starting from the `B`-th and that of `[B:L:s]` to select the elements at positions `B`, `B+s` ... `B+(L-1)*s`. The evaluations of `B`, L and `s` are unsequenced.

The ... are unsequenced. Could be omited as per 6.5.1, p.4.

3 Empty selectors and range selectors are collectively called *array selectors*. Empty selections and range selections are collectively called *array selections*.

Constraints

4 The conditional expressions shall have integer type. If the postfix expression is not an array with selection then: If the array selector has the form `[]` or `[:]` the postfix expression shall be a complete array; if it has any of the other two forms the postfix expression shall be a pointer to a complete object or an array.

5 If the postfix expression in a range selection is an array with selection its innermost selected elements shall be of array type.

Here is the restriction that in A[R][R'], A[R] cannot be an array of pointers.

6 If the array selection is not discarded, if `L` is an integer constant expression it shall be greater than zero.

7 If the array selection is not discarded and the postfix expression is of array type, any of the

expressions `B` and `B+L` in the form `[B:L]`, or `B` and `B+(L-1)*s` in the form `[B:L:s]`, if an integer constant expression, shall not be negative. If further the array is complete, let l_a be the length of the array if it does not carry a selection or the length of each innermost selected element if it carries a selection; if l_a is fixed constant, the value of those expressions as well as that of `L` in the form `[B:L]` if an integer constant expression shall be less than l_a.

Semantics

8 In the following paragraphs, let *b, l* and *s* be the value of `B`, `L` and `s` respectively. *l* shall be nonnegative. No restriction applies in general to s; in particular, it may be zero. An array of size zero (that can only be created via a range selection where *l* is zero) shall not be the operand of a **`sizeof`, `_Countof`** or typeof operator that is evaluated.

9 An array with a memory layout different from that of an array without selection is a *broken array*. A broken array is *broken*-qualified. Different broken arrays that have the same element type and number of elements may have different memory layouts. Conditions under which an array with selection is broken are stated in the following paragraphs.

10 If the expression is of the form `A[B:L]` or `A[B:L:s]` and `A` is a pointer or an array which does not carry a range selection, the first form selects the range of *l* elements starting from the *b*-th, the latter counted from zero. In the second form the expression `s` is called the *step,* as well as the value *s*; it selects the elements `A[b]`, `A[b+s]` ... `A[b+s*(l-1)]` (if *s* is zero it selects `A[B]` *l* times). The resulting array is said to *carry a selection* or to *have (a range of) elements selected*. Let *type* be the type of `A[0]` if `A` is a pointer or an lvalue, and let it be the unqualified non-atomic version of that type otherwise. The expression has type "array of *type*" and it is an lvalue if `A` is a pointer or an lvalue. It has length *l*. If `L` is an integer constant expression it is an array of fixed constant length; otherwise, it is a top-level variable length array. The elements of the resulting array preserve the order of the selection, so that, if *l* is not zero, `A[b]` is the first, `A[b+1]` or `A[b+s]` the second, etc., and `A[b+l-1]` or `A[b+s*(l-1)]` the last (if *l* is zero the resulting array has zero elements).

11 If the expression is of the form `A[B:L]` or `A[B:L:s]` and `A` is an array with a range of elements selected, let `A` have an *m*-dimensional range of elements selected of type (the innermost ones) *T*, and let *T* be array of *T'*. The expression is an array with an (*m*+1)-dimensional range of elements selected of type *T'*, obtained by selecting from each selected element *v* in the *m*-dimensional range the subarray *v*`[B:L]` or *v*`[B:L:s]`, which is as described by the previous paragraph (*v* is not an array with selection). Let *u* represent the innermost arrays in `A` that carry a range selection (these arrays' elements are the *v*'s), and let l_v be the length of the elements *v*. If $b \neq 0$ or $l \neq l_v$ the arrays *u* in `A[B:L]` or `A[B:L:s]` are broken.

12 Whenever the range selection is evaluated all the selected elements shall exist; i.e., if *l* is not zero all indices *b*, *b*+*s*, ... , *b*+(*l*-1)**s* shall be within bounds (not including one past the last element). Here *s* is 1 if the selection is of the form `[B:L]`.

13 A selection of the form `[B:L:s]` where *l* is greater than 1 and where *s* is not 1 is a *stepped selection*. An array carrying a stepped selection is broken.

14 If an array has nonzero length and its elements are broken arrays, the array itself is broken.

15 An expression of the form `A[:]` is equivalent to `A[0:_Countof(A)]`, except that `A` is evaluated only once.[xx)]

16 An empty selection is of the form `A[]`. If `A` is an array with selection the empty selection has no effect. Otherwise `A` is an array without selection and the expression is the same array with a 0-dimensional range of elements selected, namely the whole array (one element

[xx)] Because it is impossible to reselect a selected dimension, here `A` cannot have length zero.

selected), and it is said that the array has, or carries, an empty selection. An array with an empty selection is equivalent for all purposes to the array with no selection, except that it cannot be converted to a pointer.

17 NOTE: As a consequence of the rules expressed above, an array that carries a nonempty selection carries a range selection in a certain number of consecutive dimensions starting from the outermost one. In addition, an array carrying an m-dimensional selection has all the elements from its first m dimensions selected. This is so because, even if the original array might have had more elements, an array with selection consist only of the selected elements.

18 EXAMPLE 1 Consider the array object defined by the declaration

```
int x[3][5];
```

`x[1:2]` is an array of 2×5 objects and has type `int[2][5]`; it is a subobject of `x` and has a one-dimensional range of elements selected, each of which is an array of five `int`. `x[1:2][:]` is the same array and has selections in both its dimensions, resulting in a two dimensional range of elements selected, each of which is a singleton of type `int`.

19 EXAMPLE 2 After the declarations

```
int n = 3;
int x[3][5], y[3][n], z[n];
```

`x[:][0:n]` is a broken array. Its type is `int[3][3]` and is a variable length array, though not a top-level variable length array. `x[1:n-2]` is a variable length array of type `int[n-2][5]`; i.e., `int[1][5]`. `y[:]` and `y[:][0:3]` both have type `int[3][3]`, but the former is a variable length array while the latter is not. `z[0:2]` is a fixed-size array of two `int`.

`x[:][1:3]` is a broken array. `x[:][1:3][0]` has type `int[3]`, carries a one-dimensional selection and is not broken.

20 EXAMPLE 3

```
float A[10], B[10];
A[0:3] + B[9:3:-2];
```

The result is the array `{A[0]+B[9], A[1]+B[6], A[2]+B[3]}`

21 EXAMPLE 4

```
int x[3][5], y[6][20];
x[:][0:2] + y[3:3][k:2:-4]; // k is of integer type
```

Here `x[:][0:2]` and `y[3:3][k:2:-4]` are broken arrays of type `int[3][2]`. As operands of the `+` operator they undergo lvalue conversion and are no longer broken.

22 EXAMPLE 5

```
int x[3][3], y[3];
x[0:1] == y[];
```

`x[0:1]` has type `int[1][3]` and has one element selected of type `int[3]`, which is also the type of `y[]`. The former could also have been written `x[0][]`, which has type `int[3]` with one element selected of type `int[3]`.

6.5.4.4 Function calls

Constraints

1 [...]

2 The number of arguments shall agree with the number of parameters. Each argument, if it

is not an array with selection, or each of the innermost selected elements if it is so, shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

Semantics

3 A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function. The list of expressions specifies the arguments to the function.

Forward references: function declarators (6.7.7.4), function definitions (6.9.2), the **return** statement (6.8.7.5).

6.5.4.4.1 Singleton calls

Description

1 A singleton function call is a function call where no argument carries a range selection (it may carry an empty selection).

Semantics

2 An argument can be an expression of any complete object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[92)]

[...]

7 Recursive function calls are permitted, both directly and indirectly through any chain of other functions.

8 EXAMPLE In the function call

```
(*pf[f1()]) (f2(), f3() + f4())
```

the functions `f1`, `f2`, `f3`, and `f4` can be called in any order. All side effects are completed before the function pointed to by `pf[f1()]` is called.

Forward references: ~~function declarators (6.7.7.4), function definitions (6.9.2), the **return** statement (6.8.7.5),~~ simple assignment (6.5.17.2).

6.5.4.4.2 Range calls

Description

1 A range function call is a function call where at least one argument carries a range selection.

Semantics

2 The arguments that do not carry a range selection are evaluated and converted as for singleton function calls. The expressions designating the arrays with range selections are evaluated, as are the arguments of each selection (the operands within brackets). The function call is replaced by l function calls, where l is the length of the range selections of all arguments that carry a range selection (these lengths shall be equal (6.5.2)). For each of these calls the arguments are:

— For the arguments that are not arrays with selection, the already converted values.

— For arrays with selection, the value of the i-th selected element for the i-th function call, converted as for singleton function calls.

To these function calls the same procedure applies, except that expressions that have been

already evaluated are not evaluated again and arguments of array type arising from the second point in the above list are not converted to pointers.

3 The actual function calls are indeterminately sequenced. The order in which the function calls take place may be different each time the range call is reached. If at least one of the function calls does not return the behavior is undefined.

4 If the return type of the function is void, that is the type of the range call.

5 If the return type of the function is not void, the range call is an array with selection: the expression has array type and carries a selection. The type of its innermost selected elements is that of the return value of the function. Its dimensions are of the form of the selections of the arguments carrying the deepest selection, i.e., carrying a selection in the greatest number of dimensions.[xx)] If, for a given dimension, at least one of the arguments is an array carrying a selection in that dimension of fixed constant length, the corresponding length of the call expression is fixed constant.

6 EXAMPLE 1

```
Y[:] = log(tan(0.5*F[:]));
Σ[:] = sqrt(N[0:n:n+1]);
p[0:n][:] = exp(Ω[0:n][:]);
```

The last function call expression is first replaced by `n` calls of the form `exp(Ω[`*i*`][:])`. Then each of these is replaced by as many calls as the array length of `Ω[0]`, of the form `exp(Ω[`*i*`][`*j*`])`.

7 EXAMPLE 2

```
float x[3], y[3][2];
atan2(y[:][:],x[:]);
```

The range call is equivalent to the six calls (in some order)

```
atan2(y[0][0],x[0]); atan2(y[0][1],x[0]);
atan2(y[1][0],x[1]); atan2(y[1][1],x[1]);
atan2(y[2][0],x[2]); atan2(y[2][1],x[2]);
```

except that the return value is a 3 × 2 matrix and that `x`, `y` and each `x[`*i*`]` are evaluated only once.

8 EXAMPLE 3

```
func(y[0:2][n], y[2:2][5], x, z[0:m]);
```

This expression is an array of dimensions `[2][5]`, both of them fixed constant. If the value of `n` is not 5 or the value of `m` is not 2, the behavior is undefined.

9 EXAMPLE 4 In the range call of `inc`:

```
void inc(float *p, const float x){*p += x;}
float *p, A[6];
/* Assign values to p, and A */
inc(p,A[:]);
```

the value in `*p` is incremented by the sum of all the elements in `A`, in some unspecified order. The translator is not allowed to parallelize the calls if it cannot ensure that.

6.5.4.6 Postfix increment and decrement operators

Constraints

[xx)] The behavior is only defined if all selections of the same depth in the different arguments are of the same length.

1 The operand of the postfix increment or decrement operator shall have atomic, qualified or unqualified arithmetic or pointer type, or be an array with elements selected, with the innermost selected elements of any of those types, and shall be a modifiable lvalue.

2 If the left operand is an array with selection and the expression is not discarded, none of the selections it carries shall be given by a selector of the form `[B:L:s]` where `s` is an integer constant expression of value zero.

Semantics

3 If the left operand is an array with selection, whenever it is evaluated none of the selections it carries shall be a stepped selection where the step is zero.

4 If the operand is not an array with selection the *operated element*s is the operand. Otherwise, it refers to each of the innermost selected elements.

5 The result of the postfix `++` operator is the value of the operand. As a side effect, the value of the operated elements object(s) is incremented by the adjustment. See the discussions of additive operators and compound assignment for information on constraints, types, and conversions and the effects of operations on pointers. The value computation of the result for each operated element is sequenced before the side effect of updating the stored value of the said element. Both operations are unsequence with respect to the same operations on other operated elements. With respect to an indeterminately sequenced function call, the operation of postfix `++` is a single evaluation. Postfix `++` on an object with atomic type is a read-modify-write operation on each operated element with `memory_order_seq_cst` memory semantics.[97)]

6 The *adjustment* used to increment or decrement the operand is the value 1 with a type and value representation as follows:

— if the operated elements have a pointer type, the adjustment has type **`ptrdiff_t`**;

— if the operated elements have complex type, the adjustment has the corresponding real type of the operated elements;

— if the operated elements have decimal floating type, the adjustment has the same type as the operated elements with a quantum exponent of 0;

— otherwise, the adjustment has the same type as the operated elements.

7 The postfix `--` operator is analogous to the postfix `++` operator, except that the value of the operated elements is decremented by the adjustment.

6.5.5 Unary operators

6.5.5.2 Prefix increment and decrement operators

Constraints

1 The operand of the prefix increment or decrement operator shall have atomic, qualified or unqualified arithmetic or pointer type, or be an array with elements selected, with the innermost selected elements of any of those types, and shall be a modifiable lvalue.

2 If the left operand is an array with selection and the expression is not discarded, none of the selections it carries shall be given by a selector of the form `[B:L:s]` where `s` is an integer constant expression of value zero.

Semantics

3 If the left operand is an array with selection, whenever it is evaluated none of the selections it carries shall be a stepped selection where the step is zero.

4 The value of the operated elements (6.5.4.5) of the prefix `++` operator is incremented. The

result is the new value of the operated elements after incrementation. The expression `++E` is equivalent to `(E+=1)`, except that in place of 1 is the adjustment (6.5.4.5). See the discussions of additive operators and compound assignment for information on constraints, types, side effects, and conversions and the effects of operations on pointers.

6.5.5.3 Address and indirection operators

Constraints

1 The operand of the unary `&` operator shall not be an array carrying a nonempty selection and shall be either a function designator, the result of ~~a `[]`~~ an array subscripting operator or unary `*` operator, or an lvalue that designates an object that is not a bit-field and is not declared with the **`register`** storage-class specifier.

6.5.5.4 Unary arithmetic operators

Constraints

1 If the operand is not an array with selection then: The operand of the unary `+` or `-` operator shall have arithmetic type; of the `~` operator, integer type; of the `!` operator, scalar type. If the operand is an array with selection, the innermost selected elements shall have a type as constrained by the previous sentence.

[...]

5 The result of the logical negation operator `!` applied to a singleton is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. The result has type **`int`**. The expression `!E` is equivalent to `(0==E)`.

6.5.5.5 The `sizeof`, `alignof` and `_Countof` operators

Semantics

4 When **`sizeof`** is applied to an operand that has type **`char`**, **`unsigned char`**, or **`signed char`**, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result ~~is the total number of bytes in the array~~ equals the number of elements of the array times the size of each of its elements (i.e., times the result of **`sizeof`** applied to one of its elements).[103] When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding.

[...]

8 EXAMPLE 2 The **`_Countof`** operator computes the number of elements in the outermost dimension of the array:

```
double array[4][9];
static_assert(_Countof array == 4); // number of elements
static_assert(sizeof array == 4*sizeof array[0]); /* number of
      bytes in the array = 36*sizeof(double) */
```

The following equality of values always holds for arrays:

`sizeof`(array) = **`_Countof`**(array)*__`sizeof`__(array[0])

6.5.6 Cast operators

Constraints

[...]

5 If the operand is an array with selection it shall carry a selection of singletons (i.e., in as many dimensions as the array has).

Semantics

6 [...]

7 Preceding an expression by a parenthesized type name converts the value of the expression to the unqualified, non-atomic version of the named type. This construction is called a *cast*.[104] If the operand is an array which does not carry a selection, it is first converted to a pointer and the cast applies to this pointer. If the operand is an array carrying a selection of singletons the cast is applied to each of its singletons; in this case, the result is an array with the same dimensions and selection as its operand and with the type of its singletons the one resulting from the cast applied to a singleton. A cast that specifies no conversion has no effect on the type or value of an expression.

6.5.7 Multiplicative operators

Constraints

2 Either operand may be an array with selection. Its singletons shall satisfy the constraints set forth in the following paragraphs.

3 Each of the operands, if not an array with selection, shall have arithmetic type. The operands of the `%` operator shall have integer type.

4 If either operand has decimal floating type, the other operand shall not have standard floating type, or complex type.

6.5.8 Additive operators

Constraints

2 Either operand may be an array with selection. Its singletons shall satisfy the constraints set forth in the following paragraphs for operands which are not arrays.

3 For addition, when the operands are not arrays with selection, either both operands shall have arithmetic type, or one operand shall be an array or a pointer to a complete object type and the other shall have integer type. (Incrementing is equivalent to adding 1.)

4 For subtraction, when the operands are not arrays with selection, one of the following shall hold:

— both operands have arithmetic type;

— both operands are pointers to, or arrays of, qualified or unqualified versions of compatible complete object types; or

— the left operand is an array or a pointer to a complete object type and the right operand has integer type.

(Decrementing is equivalent to subtracting 1.)

5 If either operand has decimal floating type, the other operand shall not have standard floating type, or complex type.

Recommended practice

6 Implementations are encouraged to issue a diagnostic message when one operand is an array with selection and the other operand is an array that is converted to a pointer to its first element, and an array of the same type but carrying a selection could be possible in place of the latter operand.

Semantics

7 If an operand is an array without selection, the array is first converted to a pointer to its first element, thence operated as specified for pointers. If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

[...]

14 *(Change* "EXAMPLE" *to* "EXAMPLE 1"*)*

[...]

15 EXAMPLE 2 The code

```
struct stra A[10];
A[:] + 1;
```

has undefined behavior. For while `A + 1` would be valid, an array with selection is not converted to a pointer, and the selected elements of `A[:]` are not of a type that could be an operand of the addition operator.

16 EXAMPLE 3 The code

```
float *p[3], A[10];
int *q[3];
p[:] = A + q[:];
```

has defined behavior: it is equivalent to `p[0] = A+q[0], p[1] = A+q[1], p[2] = A+q[2]`.

17 EXAMPLE 4 In the code

```
int a[3], b[3];
unsigned long sum[3];
sum[:] = (unsigned long)(a + b[:]);
```

implementations are encouraged to issue a diagnostic message for the operation `a + b[:]`.

6.5.9 Bitwise shift operators

Constraints

2 Each of the operands shall have integer type or be an array with selection with its singletons of integer type.

Semantics

3 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand, when a singleton, is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

4 When `E1` and `E2` are singletons, the result `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros. If `E1` has an unsigned type, the value of the result is $E1 \times 2^{E2}$, wrapped around. If `E1` has a signed type and nonnegative value, and $E1 \times 2^{E2}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

5 When `E1` and `E2` are singletons, the result of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of $E1 / 2^{E2}$. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

6.5.10 Relational operators

Constraints

2 Either operand may be an array with selection. The innermost selected elements shall be singletons and satisfy the constraints set forth in the following paragraphs for operands which are not arrays.

3 One of the following shall hold when the operands are not arrays with selection:

— both operands have real type; or

— both operands are pointers to, or arrays of, qualified or unqualified versions of compatible object types.

4 If either operand has decimal floating type, the other operand shall not have standard floating type, or complex type.

Semantics

5 If an operand is an array without selection, the array is first converted to a pointer to its first element, thence operated as specified for pointers. If both operands have arithmetic type, the usual arithmetic conversions are performed. Positive zeros compare equal to negative zeros.

6.5.11 Equality operators

Constraints

2 If no operand is an array one of the following shall hold:

[...]

3 If either operand is an array not carrying a selection it is converted to a pointer to its first elements and it is to these pointers to which the constraint in the previous paragraph applies.

4 If either operand has decimal floating type, the other operand shall not have standard floating type, or complex type.

5 If one of the operands is an array carrying a selection, let T be the type of its singletons. Either of the following shall hold:

— The other operand is not an array with selection. If this other operand has pointer type or is an array converted to a pointer, or has type **`nullptr_t`**, T is a pointer type or **`nullptr_t`**. The type T and the other operand satisfy the constraints expressed in the previous paragraphs for the types of the operands.

— Both operands are arrays with selection. If the singletons of one of the arrays have pointer type or type **`nullptr_t`** so have the singletons of the other array. The type T and the analogously defined type T' for the other operand satisfy the constraints expressed in the previous paragraphs for the types of the operands.

Recommended practice

6 Implementations are encouraged to issue a diagnostic message when one operand is an array with selection and the other operand is an array that is converted to a pointer to its first element, and an array of the same type but carrying a selection could be possible in place of the latter operand.

Semantics

7 If an operand is an array without selection, the array is first converted to a pointer to its

first element, thence operated as specified for pointers. ~~The == (equal to) and != (not equal to) operators are analogous to the relational operators except for their lower precedence.105)~~ When operating on singletons, each operand yields 1 if the specified relation is true and 0 if it is false, the result has type **int** and exactly one of the relations is true.

and remove the footnote

8 [...]

9 ~~Otherwise, at least one operand is a pointer.~~ [...]

If both arrays carry selection, the recursive rules from 6.5.2 apply. There remain A[] *eq* s and A[] *eq* B[]

10 [...]

11 [...]

12 If one operand is an array with selection with the innermost selected elements of array type, or if it is an array with no selection which does not decay to a pointer (this is a selected array from some enclosing array), and the other operand is a singleton, the result is an array with selection unless the selection is empty or there is no selection, in which case it is a singleton. The result has one element in place of each selected array (or of the whole array if it does not carry a selection), which for the `==` operator has value 1 if all singletons of the said array compare equal to the other operand and value 0 otherwise. For the operator `!=` the values are the opposite. The resulting array carries a selection in all its dimensions.

A *eq* s
A[] *eq* s

13 If one operand is an array carrying a selection where the innermost selected elements are arrays *a*, and the other operand is an array `B` carrying an empty selection or carrying no selection and not decaying to a pointer (the latter is a selected array from some enclosing array), the dimensions of the selected arrays *a* and the array `B` are be the same (6.5.2, constraint 3). The result is an array with one element in place of each *a*, which for the `==` operator has value 1 if all singletons of *a* compare equal to the corresponding singletons of `B` and value 0 otherwise. For the operator `!=` the values are the opposite. The resulting array carries a selection in all its dimensions.

A[] *eq* B

14 EXAMPLE 1

```
int A[4][3], B[4][3], C[3];
int E[4][3], F[4], G, H[4], I[4], J;

E[:][:] = A[:][:] == B[:][:];
F[:]    =    A[:] == B[:];
G       =     A[] == B[];
H[:]    =    A[:] != 2;
I[:]    =    B[:] == C[];
J       =     A[] == 2;
```

`F[0]` equals `(A[0][0]==B[0][0] && A[0][1]==B[0][1] && A[0][2]==B[0][2])`, and similarly for `F[1]`, `F[2]` and `F[3]`. `G` is 1 if all twelve elements of `A` compare equal to the corresponding elements of `B`. `H[0]` equals `!(A[0][0]==2 && A[0][1]==2 && A[0][2]==2)`, and similarly for `H[1]`, `H[2]` and `H[3]`. Each `I[`*i*`]` equals `(B[`*i*`][0]==C[0] && B[`*i*`][1]==C[1] && B[`*i*`][2]==C[2])`. `J` is 1 if all twelve elements of `A` are 2.

15 EXAMPLE 2

```
int A[5][4][3], B[5][3], C[5][3];
int D[5][4], E[5][4];
D[:][:] = A[:][:] != B[:];
E[:] =  B[:][:] != C[:];  // Undefined behavior: E[i] = (B[i][:] != C[i])
```

Each `D[`*i*`]` is determined by the general recursive rule in 6.5.2 and equals `A[`*i*`][:] != B[`*i*`]`.

These in turn are determined by the rule of paragraph 12, so that `D[i][j]` equals `!(A[i][j][0]==B[i][0] && A[i][j][1]==B[i][1] && A[i][j][2]==B[i][2])`. The expression for `E` is, in the first place, determined also by the recursive rule, yielding `E[i] = (B[i][:] != C[i])`. This one has undefined behavior because the operand `C[i]` is an array which does not decay to a pointer and which does not carry a selection and `B[i][:]` is an array carrying a selection of singletons, and there is no rule for that case.

16 EXAMPLE 3

```
int A[1], B[1];
int C[1];

C[:] = A[:] == B[:];
C[0] =  A[] == B[];
```

The result of the first comparison is an array which is assigned to an array. The result of the second comparison is a singleton which is assigned to a singleton.

17 EXAMPLE 4 Implementations are encouraged to issue a diagnostic message for the first of the following comparisons:

```
void *A[3], *B[3];
int c;

c =  A[] == B;    //1 if each A[i] equals (void*)B.
c =  A[] == B[]; //1 if each A[i] equals the corresponding B[i].
```

The diagnostics can be avoided by writing `(void*)B` in place of `B`.

6.5.12 Bitwise AND operator

Constraints

2 Each of the operands shall have integer type or be an array with selection with it singletons of integer type.

6.5.13 Bitwise exclusive OR operator

Constraints

2 Each of the operands shall have integer type or be an array with selection with with it singletons of integer type.

6.5.14 Bitwise inclusive OR operator

Constraints

2 Each of the operands shall have integer type or be an array with selection with it singletons of integer type.

6.5.17 Conditional operator

Constraints

2 The first operand shall have scalar type.

3 If the second and third operands are singletons one of the following shall hold for them:

[...]

4 If either of the second or third operands is an array without selection it is converted to a

pointer to its first elements and it is to these pointers to which the condition in the previous paragraph applies.

5 If either of the second or third operands has decimal floating type, the other operand shall not have standard floating type, or complex type.

6 If either of the second or third operands is an array carrying a selection then so shall be the other. Their number of dimensions, considered as multidimensional arrays, shall be the same. The number of dimensions with selection in both operands shall be the same. If for any dimension the lengths in one and the other operand are given by integer constant expressions, they shall be the same. The singletons of which one and the other array are composed shall satisfy the constraints set above for operands which are not arrays. In addition, if the singletons of one of the arrays have pointer type or type **`nullptr_t`**, so shall have the singletons from the other operand.

Semantics

7 [...]

8 If the second and third operands have arithmetic type (after conversion to pointer for arrays not carrying a selection), [...].

9 If one of the second and third operands is a null pointer constant, the result has the type of the other operand. Otherwise, if one operand has type **`nullptr_t`**, the result has the type of the other operand. Otherwise, if both operands are pointers, the result type is a pointer to a type qualified with all the type qualifiers of the types referenced by both operands. If the latter types, unqualified, are compatible, the result is a pointer to the appropriately qualified version of the composite type; otherwise, one of the operands is a pointer to **`void`** or a qualified version of **`void`** and the result type is a pointer to the appropriately qualified version of **`void`**.

10 All array length expressions that are not integer constant expressions and which are part of the type of the operand that is not evaluated are treated as unspecified lengths in the determination of type compatibility and when forming the composite type according to 6.2.7.

11 If both the second and third operands are arrays with selection, the conditional expression carries the selection of the evaluated operand. The type of its singletons is determined by applying the above rules to the types of the singletons of both operands.

6.5.18 Assignment operators

6.5.18.1 General

Constraints

2 An assignment operator shall have a modifiable lvalue as its left operand.

3 If the left operand is an array with selection and the expression is not discarded, none of the selections it carries shall be given by a selector of the form `[B:L:s]` where **`s`** is an integer constant expression of value zero.

Semantics

4 [...]

5 If the left operand is an array with selection, whenever it is evaluated none of the selections it carries shall be a stepped selection where the step is zero.

6.5.18.2 Simple assignment

Constraints

1 If the left operand is not an array one of the following shall hold:

[...]

2 If the left operand is an array, its singletons together with the right operand if not an array with selection or the singletons of the right operand if it is an array with selection shall satisfy the constraint above for the case the left operand is not an array. Furthermore, if the right operand is an array with selection and the singletons of the left operand have pointer type or type **`nullptr_t`**, so shall have the singletons of the right operand.

3 If the right operand is an array with selection the left operand shall be an array. The number of dimensions with selection in the left operand shall be greater than or equal to the number of dimensions with selection in the right operand (an empty selection is considered a selection in zero dimensions).

Semantics

4 In *simple assignment* (`=`), ...

5 If the left operand is not an array and ~~if~~ the value being stored in ~~an object~~it is read from another object that overlaps in any way its storage, then the two objects shall occupy exactly the same storage and the type of the expression used to access the object read shall be a qualified or unqualified version of a type compatible to that of the left operand; otherwise, the behavior is undefined.

7 If the left operand is an array, for each singleton i of it in which a value is stored by the assignment let C(i) be the set of its singletons that need to be read, in whole or in part, in the expression at the right of the assignment operator in order to compute the value to store in i. If C(i) includes another singleton of the array which also has a value stored in it by the assignment, the behavior is undefined. If C(i) includes i, the condition in the previous paragraph applies to it. C(i) includes all the values that are read in the abstract machine in the chain of operations expressed by the right operand that determine de value to store in i, even if they are not needed from the mathematical viewpoint.

Recommended practice

8 Implementations are encouraged to issue a diagnostic message when the left operand is an array of pointers, the right operand is an array of pointers that is converted to a pointer and an array equal to the right operand but not decaying to a pointer would also be possible as the right operand.

[...]

12 EXAMPLE 4 Consider the following assignments

```
int A[6];
A[1:3] = A[1:3] + 0*A[3];
A[:] = A[:] - A[:];
A[:] = A[3];
```

In the first assignment C(i) is {i, `A[3]`} for each i (where `A[3]` means the object denoted by `A[3]`, not the value of the expression `A[3]`) and the behavior is undefined. In the second assignment C(i) is {i} for all i. In the third assignment C(i) equals {`A[3]`} for all i and the behavior is also undefined, even if the expression does not change the value of `A[3]`.

13 EXAMPLE 5 In the following piece of code

```
int A[9];
char B[7];
A[0:3] = A[B[0]:3:-1];
```

```
A[3] = ((char*)A)[0:7] == B[];
A[3:3] = ((char*)A)[0:7] == B[];
```

the first assignment has defined behavior if $4 \le$ `B[0]` ≤ 8. The second assignment has defined behavior always because the left operand is not an array and the value to store in it is not read from an object, while the third one has undefined behavior if **`sizeof(int)`** ≤ 2.

[...]

14 EXAMPLE 6 Assignments where the left operand is an array.

```
int A[5][4], B[5][4], C[4];
int *D[12];
A[:] = C[];             // Equivalent to A[0][:]=C[:], A[1][:]=C[:], etc.
A[:] = B[0][];          // A[0][:]=B[0][:], A[1][:]=B[0][:], etc.
D[:] = B[2];            // D[i] = &B[2][0]
// The following four assignments are equivalent
A=B[]; A[]=B[]; A[:]=B[:]; A[:][:]=B[:][:];
```

In the first and second assignments above, writing `C[]` and `B[0][]` instead of `C` or `B[0]` is necessary to prevent the array from decaying to a pointer.

15 EXAMPLE 7 Implementations are encouraged to issue a diagnostic message in the following assignment:

```
void **A, *B[3];
A[0:3] = B;
```

6.5.18.3 Compound assignment

Constraints

1 Either operand may be an array with selection. The left operand may be an array with no selection. In these cases, it is the singletons of the arrays that shall satisfy the constraints set forth in the following paragraphs.

2 For the operators `+=` and `-=` only, if the operands are not arrays then: either the left operand [...]

3 For the other operators, if the operand are not arrays then: the left operand [...]

4 If either operand has decimal floating type, the other operand shall not have standard floating type, or complex type.

6.7.3.6 typeof specifiers

Constraints

3 The typeof operators shall not be applied to an expression that designates a bit-field member.

4 The **`typeof`** operator shall not be applied to an array carrying a nonempty selection.

6.7.7.3 Array declarators

[166]The declaration of variable length arrays with automatic storage duration is a conditional feature that implementations are not required to support; see 6.10.10.4.

6.7.10 Type inference

Semantics

2 For such a declaration that is the definition of an object the init-declarator shall have the form

direct-declarator `=` *assignment-expression*

The inferred type of the declared object is the type of the assignment expression after lvalue, array to pointer or function to pointer conversion, additionally qualified by qualifiers and amended by attributes as they appear in the declaration specifiers.[169)] The assignment expression shall not be an array of size zero.

6.10.10.4 Conditional feature macros

`__STDC_NO_VLA__` The integer literal `1`, intended to indicate that the implementation does not support the declaration of variable length arrays with automatic storage duration. Parameters declared with variable length array types are adjusted and then define objects of automatic storage duration with pointer types. Thus, support for such declarations is mandatory.

`__STDC_RANGE_SELECTIONS__` The integer literal `1`, intended to indicate that the implementation supports array selections beyond empty selections. Otherwise the macro shall not be defined or defined to `0`.

If the implementation supports only empty selections, the following two macros may not be defined; if defined, the first one shall be defined to `0`.

`__STDC_ARRSEL_NESTED__` The integer literal `0` if selections of the form `[B:L]` and `[B:L:s]` can only be applied to expressions of pointer type and to arrays carrying no selection or an empty selection, and the type from which the pointer type or the array type is derived is not an array type. The integer literal `1` otherwise.

`__STDC_ARRSEL_STEPPED__` The integer literal `0` if selections of the form `[B:L:s]` are not supported; the integer literal `1` if they are.