Proposal for C

Array Notation for Vectorization

Javier A. Múgica

January 28th , 2025

This extension was suggested to the author by **M. Uecker,** from whom I received valuable suggestions in the early stages of the design

Table of contents

A[B:-L]

Range selection constraint related recommended practice

Relaxing the UB of overlapping in assignments

Relaxing the restriction for overlapping

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

typeof, sizeof, _Unselect() and _Value()

Functions taking and returning arrays

Functions acting as range operators

Address to broken arrays. The broken qualifier

13 Further editorial fixes

6.5.16 Conditional operator

6.5.17 Assignment operators

6.5.17 Assignment operators (again)

14 Wording

6.2.5 Types

6.3.3.1 Lvalues, arrays, and function designators

6.5 Expressions

6.5.2 Arrays with selection as operators

6.5.4 Postfix operators

6.5.5 Unary operators

6.5.6 Cast operators

6.5.7 Multiplicative operators

6.5.8 Additive operators

6.5.9 Bitwise shift operators

6.5.10 Relational operators

6.5.11 Equality operators

6.5.12 Bitwise AND operator

6.5.13 Bitwise exclusive OR operator

6.5.14 Bitwise inclusive OR operator

6.5.17 Conditional operator

6.5.18 Assignment operators

6.7.3.6 typeof specifiers

6.7.3.6 Typeof specifiers

6.7.7.3 Array declarators

6.10.10.4 Conditional feature macros

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[:][0:6] += A[:][6:6];
A[b:n] = -B[0:n];
A[0:5:2] = 1/A[0:5:2];   // Operate on the elements 0, 2, 4, 6, 8
```

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 arrays as arguments. This is possible because we have chosen arrays with selected elements not to decay to pointers. This is explored in "Further extensions".

Prior art

The specification of a range in the index or a comma-separated list of values is common in many languages. Some languages specify begin and end instead of begin and length. In C, the former seems preferable, 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]`.

Intent of the proposal

In a first step of the design we only defined continuous selections, and only one-dimensional. Stepped selections and multidimensional selections are a natural extension. This leads to a more complicated proposal. Not only the needed changes to the standard become longer, but the consideration of contrived examples grows considerably. Any design has to take into consideration any possible combination. These are thoroughly discussed here. In several cases the conclusion is that no extra wording is needed, but the study of the particular case is necessary nonetheless.

After these kinds of selections we considered the ones we call indexed selections and direct selections, which need yet further wording and consideration of cases. For these two we do not propose a concrete wording, though they are analysed to a point that brings them near wording.

Small implementations are reluctant to adopt complex features, and there is already a not so short list of optional features: variable length arrays, complex numbers, decimal floating numbers, atomics, threads, to which may be added recent additions to the language that some implementations do not plan to support in the foreseeable future. Some prefer to keep the language small and avoid any complex features, whether made optional or not. But at the same time implementations cannot be stopped (and should not be) from

providing extensions, and it is good for the programming community if the way these extension are provided is standardised. For this reason we believe that the full set of features with respect to empty selection and range selections should be put in a standard form. Other features analysed in this paper go too far afield. We also believe that implementations should not be forced to implement the most complex combinations, or range selections at all.

We don't intend the wording proposed here to be voted for inclusion in the standard now. 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.

Therefore, an actual wording is desirable. The committee should express its intent in adopting the feature in some concrete from, the one proposed here or another one, but a concrete one, so that implementations can embark in the task of implementing it with the assurance that they are implementing the design that will be adopted by the standard. In doing this they may detect issues with the proposed wording that need adjustment, and the wording eventually comes to a final form. This way implementers see that they guide the shaping of the final wording, not that the final wording makes their implementation incompatible with the standard.

Another reason for developing the feature in full is that the analysis of more complicated cases helps in the design of the simpler ones. 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 that it is by then too late to change.

This document, therefore, may serve as a basis for a technical specification or simply for a proposal available from the project page, to which the committee has expressed its approval but the wording of which may not yet be in final form and bears no normative power.

Once implementations begin implementing this, as we hope, the adoption into the wording of the standard may be done progressively, incorporating first single-dimensional, continuous (i.e., no step) selections, for example. Since the feature is likely to remain optional it may go into an annex (except empty selections).

Another feature that can be incorporated easily is the empty selection: `A[]`, which essentially serves to suppress conversion to pointer of the matrix. We don't intend this to be optional. The adoption of empty selections make very easy the introduction of functions that can take arrays as arguments, which may be the next addition in this line.

Relation to other proposals

Range operations on arrays lead naturally to an empty selection for selecting the matrix as a whole and to matrices not decaying to pointers. The syntax that arises naturally for this should be taken into consideration by others thinking of a way of avoiding arrays decaying to pointer in some contexts; for instance, for functions taking arrays.

The proposal of fixing the order of evaluation in operations like `A+B` has to consider very carefully its implications for range operations, where one obvious translation consists in first evaluating all the expressions needed for determining all the ranges that appear in the expression (e.g., in an assignment expression), in our notation, `B`,`L`,`s`, then translate the range operation as a sequence of individual operations, as though it were a **`for`** loop.

The terminology with respect to the number of elements and size of an array has to be very precise. We use the following terms: *length* to mean the number of elements. *Total*

length is the product of the length of the array times the length of its elements and so forth; that is, the total length of an array is its length if its elements are not of array type, the product of its length times the total length of its elements otherwise. *Variable length array,* with its usual meaning, even though the term corresponding to that meaning should be variable size array. Its opposite is array of *known constant size*. Since there do not exist singletons (the term singleton is introduced soon in the proposal) of variable size, known constant size coincides with *known constant total length*. This latter term is used only once, at a point where it is important to emphasize that the text refers to the total number of elements, not the size. The opposite of this term should be variable length array, had not this term the meaning it has. If the length of the array is known (i.e., given by an integer constant expression), it is an array of *known constant length*; otherwise it is a *top-level variable length array*. We also use the term *fixed length* as a synonym of known constant length, but not in the wording.

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 `A[B:L]` denotes, or *selects*, the subarray of `L` elements `A[B]` ... `A[B + L−1]` and `A[:]` 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 the 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 this is the combination of selections with the array subscripting operator, `[`k`]`.

Singleton

Before anything else, we introduce the concept of singleton. For a time it appeared that we could get along with the expression “element of non-array type”, but in the end it proved impossible in practice. As way of example, this document contains well over one hundred uses of the term. The definition we provide for it is as follows:

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.*

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

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, or rather a selection thereof, 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, which the expression `A` does not.

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][0:5]
```

denotes an array of four elements, also of type "array of five int". The former has its three elements selected, which are of type "array of five int" while the latter has twelve elements selected in a 4 x 3 array, which have type "int".

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 l-di-

mensional array of m-dimensional arrays, and $l+m = n$. 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 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 l-values:

```
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. Long and short interpretations

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 s denotes an element that has been selected and will be operated and i an element which will be ignored:

$s, s, s, i, i, s, s, s, i, i, s, s, s, i, i, s, s, s, i, i$

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*; and each dimension with a selection in which not all the elements are selected, we will call a *broken dimension*. 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.

New type

These broken arrays made us think for a moment to enlarge the type system for them,

based on the principle that one of the characteristics of a type is the layout of its elements. Thus, each broken array would be akin to a structure, broken arrays differing in the number of elements in each dimension or the selected elements from each of these being of different type. For example, `A[:][0:2]` and `A[:][2:2]` would be of different types. Apart from the enlargement of the type system, that we prefer to avoid, this choice presents another problem: what would be the type of `A[:][0:2] + A[:][2:2]`? This latter is readily solved by the way we specify lvalue conversions of arrays with selections. We still prefer not to enlarge the type system for arrays with selections.

Long interpretation

We cannot naïvely assign `A[2:4][0:3]` above type **`int`**`[4][3]`, since the memory layout of the object is not that of an **`int`**`[4][3]`. It is that of an **`int`**`[10][5]`, with some elements selected, others ignored. With this interpretation, each broken array has a corresponding *full array,* which is the array from which it is a selection, with no selection.

This choice poses a dilemma when the range `[B:L]` is selected from the outermost dimension of a declared array `A`. For symmetry with inner dimensions we would say that `A[B:L]` is a broken array, but this contradicts what said at the onset of the *Continuous subarray selection* section about the type we want for `A[B:L]` and leads to some unexpected semantics, as in

```
int A[10];
sizeof A[0:1]; //Would equal 10*sizeof(int)
```

Furthermore, the selection `[B:L]` may be applied to a pointer, in which case the result must necessarily be an array of `L` elements.

For these reasons we prefer to let the type of `A[B:L]`, when A is an array which does not carry a selection, be that of the sliced down array; i.e., the same as that of `A` except possibly for the number of elements:

```
int A[10][5];
A[2:4];      //Same type as int[4][5]. All its four elements A[i] are selected.
A[2:4][0:3]; //The type is int[4][5]. The selection is a broken array.
A[2:4][:]; //The type is int[4][5]. All its 20 elements A[i][j] are selected.
```

All three objects in the example have the same size; namely, that of an **`int`**`[4][5]`.

The main drawback of the long interpretation is that the type of a broken array changes upon lvalue conversion. Thus, `A[2:4][0:3]` is an **`int`**`[4][5]` before lvalue conversion and an **`int`**`[4][3]` after it. This makes the use of broken arrays dangerous as operands to **`sizeof`**, **`typeof`** and other places.

Short interpretation

As already pointed, we cannot just say that the type of `A[2:4][0:3]` is **`int`**`[4][3]`. Yet, that would make the type of that array the same before and after lvalue conversion. The solution is to attach to it a qualifier. 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 atomic. 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, the type of the object is the same. We will discuss at the end of the analysis, just before the wording, whether we want broken arrays in the short interpretation to be atomic-like (qualifier) or register-like (storage class).

The type of the arrays considered above is now

```
int A[10][5];
A[2:4];      //Same type as int[4][5]
A[2:4][0:3]; //The type is int[4][broken 3]
A[2:4][:]; //The type is int[4][5]
```

The word `broken` is not a keyword proposed, just a symbolic way of representing the type of broken arrays in the short interpretation.

Here there is no doubt that the type of `A[B:L]` is `int[L]`. It is not `broken` because it is not broken in the literal sense; i.e., its elements are not spread out.

In the short interpretation, whenever a dimension carries a selection, all the elements of the array which carries the selection are selected; the elements of the original array that were not selected do not form part of the array after selection.

The choice taken

In the end we chose the short interpretation. All this document considers both the long and short interpretations. This is for one reason: in order to assess the two options the implications for the different operands have to be analysed in detail. The long interpretation presents more nuances than the short one, which is more straightforward. It presents, for instance, the dichotomy of margins or not margins (to be explained later), the interpretation of array subscripting, etc. After pondering the drawbacks of both interpretations, we choose the short one. The proposed wording, accordingly, is for the short interpretation.

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

A variable length array

If `L` is not a constant expression a fixed length array could be turned into a variable length array, and vice-versa:

```
int n = 4;
int A[10], B[n];
A[0:n]; //Variable length array
B[1:2]; //Array of known constant size
```

We should consider with respect to this problem what should be the semantics when `A` is a pointer, not an array, as in

```
void mult_array(float *A, float *B, float *C, int n){
      A[0:n] = B[0:n] * C[0:n];
}
```

It seems difficult to make `A[0:n]` have a type other than variable length array of length `n`.

All this leads us to let the type of `A[B:L]` when `A` does not carry a selection be `typeof(A[0])[L]`.

When `A` does carry a selection, the type is be given by the length of `A` in the long interpretation; i.e., it is the same as that of `A`, while in the short interpretation it is always `typeof(A[0])[L]`, and may or may not be broken.

Mandatory VLA

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 be henceforth 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. But given that there are no other instances, the first wording makes it easier for a reader to know exactly what support for VLA is not mandatory. Therefore, we took the first approach.

The selection [::]

Consider the following code:

```
int A[3][3], B[3][3], C[3][3];
A[:][:] = B[:][:]+C[:][:];
```

It may seem more natural to write

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

A “partial selection”, in which there are not as many `[:]` or `[B:L]` as possible seems of very limited use. For this reason it was considered to make a `[:]` selection operate on all the dimensions, except that if selections `[B:L]` follow, these refer to the innermost dimensions and the `[:]` would apply to the remaining ones. There should not be more than one `[:]` in a series of consecutive selection operators.

However, others may prefer not to have a special rule for `[:]`. Furthermore, for the operators `==` and `!=` these partial selections are useful.

Both sensibilities can be reconciled by devising a different syntax for the selection of

the whole matrix. We chose `[::]` for it:

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

For a time we relegated this to “future extensions”, but it proved so useful when working with multidimensional arrays that we included it in the main proposal. We found ourselves using it in, e.g.,

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

This has the advantage that the programmer need not care about the number of dimensions of the matrices involved. The above statement is written the same way whether `A`, `B` and `C` are unidimensional, bidimensional, etc. Of course, the dimensions of the three matrices must match. This can be very useful for macros.

Also, it expresses intent better than

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

Here, the reader of this code cannot know just by looking at it if the comparison is carried at the level of the singletons or if the matrices are multidimensional and the comparison is between vectors or matrices (see below the discussion for the `==` operator).

As noted above, we make `[::]` select all dimensions up to the following `[:]`, `[B:L]` and `[B:L:s]` selections if present (see the wording).

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 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; in particular, lvalue conversion transforms a stepped selection into a continuous array. The exception is a stepped selection as the first, outermost one. This creates from the outset an array with its first dimension broken.

Long interpretation

If `A` is an array which does not carry a selection, `s` is an ICE and either `s` is zero or `L` is an ICE (or both)

`A[B:L:s]` has type `int[s*(L-1)+1]` (for `s`≥0).

The choice `s*L` is also natural, each selected element being followed by `s-1` padding elements. But this is not possible for cases like `int A[7]; A[0:3:3]`, where the last selected element is also the last element in the array. Nor is it possible if `s` is zero. If `s` is negative, it is its absolute value which is taken.

Both interpretations

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. In the long interpretation, an array of one element results if `[B:L:0]` is the outermost selection. In the short interpretation the resulting array always has `L` elements, even if they will share the same location in memory.

Finally, `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:

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

Long interpretation

If the element type of `A` is **`int`** (or `A` is a pointer to **`int`**), `A[B:L:0]`has type **`int`**`[1]` with no ignored elements. It is not a broken array.

A corner case arises if `A[B:L:s]` is an array of known constant size, `s` is an ICE which evaluates to zero, `L` is not an ICE and `A[B:L:s]` undergoes lvalue conversion. Before lvalue conversion the array has size 1 and is therefore of known constant size. After the conversion it has `L` elements, hence is not of known constant size. This is the only case where lvalue conversion changes an array of known constant size into a VLA. We think this is not problematic since the translator will handle the array after conversion in either of two ways:

— It copies the selected values in a temporary object in memory, the amount of copies is not known at translation time. This happens already (in this proposal) for any array with selection which is not of known constant size.

— The elements are operated with some other element or the elements of another array with selection; these operations are translated as a **`for`** loop. Then, the fact that `s` is zero is irrelevant: each iteration of the **`for`** loop reads the required value, be it always the same or not. We *do not* allow constructions where the repeated element would be the destination of writing operations.

It would be problematic for unary operators, as in `A[0:L:0]++`. But here the problem arises independent of `L` not being a constant expression. It is the one just treated above, which is solved by not allowing stepped selections where `s` is zero as the left operand of an assignment operator.

lvalue conversion of an array with a stepped selection where s is zero and l is not one constitutes an exception, in that the array after lvalue conversion has more elements than before; in any other situation is has the same or less elements.

Short interpretation

Now the type of `A[B:L:0]` is **`int`**`[`l`]`, even before lvalue conversion. 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` qualifier for this. Thus, *boken* does not necessarily mean spread out, but *having a layout different from the unqualified array;* typically because its elements are spread out, but can also be the opposite: different elements sharing the same space in memory.

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, the translator must evaluate `B`, `L` and `s` before translating the range operation proper, so side effects take place naturally at that point. 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.

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

General rule for binary operators

Whenever an expression is of the form `A[R]`... *op* `B[R']`... , for most operators *op*, where `[R]` and `[R']` are range selections other than `[::]`, the number of selected elements in `[R]` and `[R']` have to be the same, call it *n*, and the expression is equivalent to

`A[`*i*`][]`... *op* `B[`*i*`][]`..., $0 \le i < n$

The presence of the empty `[]` will be explained in "0-dimensional selection", and prevents the array to its left to decay to a pointer. and `A[`*i*`]` and `B[`*i*`]` run through the selected elements. If a range selection is `[::]` is first replaced by the equivalent series of `[:]`, then the rule is applied. For example,

```
int A[6][6], B[6][6];
A[::] + B[::]; // A[0][:]=B[0][:], etc.
A[:] == B[:];  // A[0]==B[0], etc., but A[0], B[0] are not converted to pointers
```

The operators of this form which are excluded from this rule are the logical 'and' and 'or' operators and the comma operator.

The application of the rule gives a way to pull the vector operation to an outer dimension in one of the operands, as in this example:

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

The rule applies to `B[3]` and `C` giving `B[3][`*i*`]*C[`*i*`][4]`, instead of, say, `B[`*i*`][:]*C[`*i*`][:]`.

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 (to be explained below). 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] // {i, A[5][1], A[5][2], i, i, i}
```

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

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

In the last example `A[5:4][1:2][0]` is a unidimensional array, yet that dimension is broken in the long interpretation. This cannot happen as a result of selecting elements from an array with `[B:L]` or `[:]` selections, the first dimension is never broken in those cases; nor is it in the short interpretation, where `A[5:4][1:2]` has type `int[4][2]`, hence `A[5:4][1:2][0]` has type `int[2]`. We consider below in the section “Margins” another possibility for `A[5:4][1:2][0]` in the long interpretation.

Array subscripting when the first dimension is broken

Long interpretation

Consider the following examples:

```
int A[9][6];
A[0:3:2][1];         // A[1] or A[2]?
A[5][1:2];           // Array of type int[2]: {A[5][1], A[5][2]}
A[5:4][1:2][0];      // {i, A[5][1], A[5][2], i, i, i}
A[5:4][1:2][0][1];  // A[5][1] or A[5][2]?
A[5:4][1:2][0][0];  // Wrong, it would be A[5][0], which is amongst the ignored
                       elements? Or A[5][1]?
```

We said that `A[5:4][1:2][0]` has type `int[6]`. So, a further `[0]` should refer to its

first element. Alternatively, we may posit that the array subscripting applied to an array with a selection operates on the selected elements. That would break the equivalence between `[0]` and `*`, unless the action of `*` is also redefined. We do not consider the latter possible: the address of and object should be that of its first byte. On the other hand, the very way in which the selection `[B:L]` is defined points towards an index in `[`k`]` being counted from the first selected elements. Considering this, three options are possible:

Forbid the `[`k`]` operator on arrays where the first dimension is broken.

The index in `[`k`]` counts from `B` and steps `s`. `*` selects the first element.

The index in `[`k`]` counts from `B` and steps `s`. `*` is forbidden on arrays with selection.

We finally went for the third option. `*` operates on pointers, while the purpose of a selection in an array is to denote the array itself (restricted to the selected elements), a range of elements on which an operand will operate; it cannot be a pointer.

Short interpretation

Now the situation is simpler because the broken array only carries with it the selected elements. We could even allow `*`, but we prefer not to allow it, for the reason just expressed.

Margins

All this section appertains only to the long interpretation, except when otherwise noted.

A broken outermost dimension can arise because of two reasons. One is a stepped selection, the other the combination of range selection and array subscripting:

```
int A[20][6], n;
A[5:4:3];
A[9:4:n];
A[5:4][1:2][0];
```

This latter only generates a broken outermost dimension in the long interpretation.

Mixing of range selection and subscripting

Consider this case first. `A[5:4][1:2]` is composed of four pieces:

{i, `A[5][1]`, `A[5][2]`, i, i, i} {i, `A[6][1]`, `A[6][2]`, i, i, i}
{i, `A[7][1]`, `A[7][2]`, i, i, i} {i, `A[8][1]`, `A[8][2]`, i, i, i}

so it seems the most natural that `A[5:4][1:2][0]` should be the first of these pieces. This results in unselected elements at the ends (margins).

These margins can be eliminated in the obvious way: letting `A[5:4][1:2][0]` be

`{A[5][1], A[5][2]}`

This goes against the most basic rules of arrays. First, an array should be the composition of its elements. Secondly, its number of elements should be **`sizeof`**`(A[5:4][1:2])/` **`sizeof`**`(A[5:4][1:2][0])`. It has the advantage that the following two expressions are equivalent:

`A[5:4][1:2][0]` `A[5][1:2]`

Some drawbacks of this interpretation disappear if we think of the operation `[0]` not as se-

lecting the 0-th element of the array with selection to its left, but as the selected elements therein. Thus, `[k]`, when applied to an array carrying a selection, selects the selected elements from the array's *k*-th element if these elements are of array type and carry a selection, or the *k*-th element if they are singletons or arrays which do not carry a selection. This way `A[5:4][1:2]` is still the juxtaposition of its four elements and each of them has a size of `5*sizeof(int)`.

With this interpretation we have `sizeof(A[5:4][1:2][0])` = `2*sizeof(int)` because `[0]` does not select the whole element. (But we are not allowing `sizeof` on arrays which carry a broken selection).

Another advantage of this interpretation is that `typeof(A[5:4][1:2][0])` is `int[2]`, which is the more natural choice given that `A[5:4][1:2][0]` has only two elements that can be addressed with no holes in-between; namely, `[0]` and `[1]` (But we are not allowing `typeof` here either).

It has the drawback that `[k]` does not select the array's *k*-th element. We may think of

`A[R][R']...[k][k']...`

where `[R]`, `[R']`, ... are range selections, of which only the first one need be present, *b* is the first element in the selection `R` and `[k]`, `[k']`, ... are array subscriptings of which only the first one need be present, as a longhand for

`A[b+k][R']...[k']...`,

except that if `[R]` is `[::]` it is first replaced by the equivalent number of `[:]` selections. But this still leaves one uneasy.

Stepped selections

In this case there will be holes between the selected elements, if the step is >1 or <-1, but there will not be empty margins. For example, the layout of `A[5:4:3]` is

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

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

Long interpretation

The layout of `A[10:4:s]` if *s*≠0 is

{*s*, *i*.(*s*-1), *s*, *i*.(*s*-1), *s*, *i*.(*s*-1), *s*}

where *i*.(*s*-1) means *s*-1 ignored elements (If *s*=0 the layout is {*s*}). But, which *s* among these four is `A[10]`? The first one or the last one? In the long interpretation it depends on whether *s* is positive or negative.

This seemed at first problematic. An object should have a well-defined address and proceed forward in memory. A variable length array can lead to a similar situation, as in

```
int A[3][n];
```

where the address of `A[1]` is `(int*)A+n`. But in this case the translator knows that the object `A[1]` starts behind of `A` and that its first element is `A[1][0]`.

This made us consider the following design for the long interpretation: In a stepped selection where the step is not an ICE the expression (the stepped selection) consists of all the elements of the array. For example, in the selection `A[10:4:s]`, with `s` not an ICE and supposing it evaluates to -3 when evaluated, the layout would be:

$\{i,\ \text{𝔰},\ i,\ i,\ \text{𝔰},\ i,\ i,\ \text{𝔰},\ i,\ i,\ \text{𝔰},\ i,\ i,\ i,\ i,\ i,\ i,\ i,\ i,\ i,\ i\}$

This way the translator always knows the size and address of the array with stepped selection.

Another consequence of this design is that if `A` has pointer type and *s* is negative it has to be given by an ICE. Or as it would be written: If `A` has pointer type and `s` is not an integer constant expression *s* shall not be negative. So that in `A[B:L:s]` the translator can know that the selection will span the positions

from `B` *to* `B + (L-1)*s`

We can form the expression `B + (s<0)*(L-1)*s` that gives unconditionally the lowest address element of the selection, but the compiler cannot know whether this lies ahead of `A+B` or behind, contrary to VLA, where it always knows it lies behind.

But since `B` itself may not be an ICE it turns out that stepped selections with a negative step are no more complicated than selections without step can be, as regards the size and the position of the lowest address selected element:

`A[B:L:s]`

`A[B':L']` *as in* `A[ B+(s<0)*(L-1)*s : 1+(L-1)*abs(s) ]`

Since in the second case we let `A[B':L']` span only the selection, from `B'` to `B' + L'-1`, there is not reason in `A[B:L:s]` not to do the same. Indeed, it seems we should do the same.

Therefore, a selection `A[B:L:s]` will always be as follows: It span from `A` + *b* + (*s*<0)*(*l*-1)**s* to `A` + *b* + (*s*<0)*(*l*-1)**s* + (*l*-1)*|*s*| = `A` + *b* + (*s*>0)*(*l*-1)**s*. And there is no restriction on `s` when `A` is of pointer type.

Short interpretation

The layout of `A[10:4:s]` can be represented as {𝔰, ? , 𝔰, ? , 𝔰, ?, 𝔰} and `A[10]` is always the first 𝔰. In a selection `A[B:L:s]`, its first element is the one at `A` + *b*; its last element, the one at `A` + *b* + (*l*-1)**s*, whether *s* be positive or negative.

Both interpretations

All the above applies before lvalue conversion. After lvalue conversion the array has the *l* elements in order, and there is nothing to say about its memory layout, for it is, conceptually, just a value.

Both constructions

(This continues to apply only to the long interpretation)

In order to assess the different options (margins or not margins) we have to ask ourselves when does the difference matter. If the expression is undergoing lvalue conversion the distinction is irrelevant. If it is the left operand of an assignment, the type given to it is also irrelevant (and the expression itself undergoes lvalue conversion). It turns out that the contexts where it matters have been excluded in this proposal: **`sizeof`**, **`typeof`**, taking its address, decaying to pointer. Another situation, what element is selected by the `[`*k*`]` array subscripting, has been settled by making the index count only selected elements, irrespective of brokenness: margins and holes.

Whatever the criterion chosen it has to be uniform. We envisage three possibilities with respect to `A[R]`, where `A` is a pointer or an array not carrying a selection:

— Never trim margins. The type of `A[R]` is always that of `A`

— Trim margins if the selection is given by ICEs, keep them if there is even one not-ICE

— Always drop margins. `A[R]` spans from the first to the last of the selected elements

For `B[`k`]`, where `B` carries a multidimensional selection, we see three possibilities, in partial correspondence with the previous ones:

— Never trim margins. The type of `B[`k`]` is the element type of `B`

— Trim margins if the selection is given by ICEs, keep them if there is even one not-ICE

— Always drop margins. `B[`k`]` spans from the first to the last of the selected elements

The option chosen for `B[`k`]` has to be ≤ the option chosen for `A[R]`. For example, if the second option is chosen for `A[R]` only the first or second options can be chosen for `B[`k`]`.

Giving `A[R]` the same type as `A` when the latter is a pointer is strange. We finally chose the third option with respect to `A`, as explained above for stepped selections. While it may require run-time computations, the situations where they are needed would happen seldom, this is already the situation for VLA, and precisely in those contexts, and the optionality provided via the macros `__STDC_ARRSEL_` allows an implementation not to handle those cases at all if it so wishes.

As for `B[`k`]`, we finally chose that it be the whole k-th selected element of `B`, including margins if there are. This is the most coherent; margins will be dropped at lvalue conversion, together with all other unselected elements, and it does not affect the meaning of a subsequent `[`k`]`, which will always be the k-th selected element in `B[0]`.

We have therefore made up a decision for the long interpretation. For the short interpretation, which we finally take, the situation is much simpler.

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];
```

As with any other pointer to an array element, this pointer can only be 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 allow 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., if the whole instruction is placed before in the generated code).

A pointer taken from an array carrying a step-zero selection will modify all the elements when modifying one of them. But this has not importance; those arrays are not allowed as the left operands of an assignment, and when undergoing lvalue conversion, the meaning for the translator is just “take element at *b*, *l* times”:

```
int A[6], B[6];
int *q= &A[3:6:0];  // q is &A[3]
q[0]=5;  // Modifies A[3]
q[1]=1;  // Invalid
B[:] = A[3:6:0];  // Could have been written more clearly as B[:]=A[3].
```

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
```

Meaning of broken in the long interpretation

The meaning is clear: the array has less elements selected than elements has. With this definition, the concept of broken dimention is precise. For example:

```
int A[6][6], B[6];
A[:][3:3];
B[6:6:-1];
```

The matrix `A[:][3:3]` has its first dimension full (unbroken) and its second dimension broken. `B[6:6:-1]` is not broken; `B[2:`*l*`:0]` is, for any *l*.

Meaning of broken in the short interpretation

Here the property of being broken acts as a qualifier, meaning that the layout is different than that of a plain array. Therefore, `B[6:6:-1]` as above has to be considered broken, as well as `int C[1]; C[0:`*l*`:1]`, for any *l* greater than one.

Consider now the selection on the matrix `A` in

```
int A[6][6];
A[:][3:3];
```

Each of the elements `A[0]`... `A[5]` is an array of three elements, of type `int[3]`, which is not broken (and which carries a selection in its first and only dimension). The matrix `A` in `A[:]` is also not broken, yet `A[:][3:3]` is broken. We cannot say that its second dimension is broken since the elements `A[`*i*`]` are not broken. We might say that its first dimension is broken, for `A[1]` does not come right after `A[0]`. But since the selection in the first dimension is `[:]`, what does it mean in the short interpretation to say that a dimension is broken?

In the short interpretation, identifying which dimensions are broken in a broken array ceases to be meaningful; the array is broken as a whole. Not beimg broken means that the elemtns are sotred one after the other, with no wholes. But if the elements are broken arrays, whose elements are spread out in memory, what does it mean to say that they are

stored one after the other? A broken array hasn't got a well defined address and extent in memory, in general. Hence, once an element of array type is broken, the array of which it is an element is broken as a whole. Therefore, being broken implies being broken in the first dimension, which is to say that the dimensions which are broken add not meaning beyond being broken: In the long interpretation it is possible that a broken array is broken in its second dimension but not in the first one; in the short intepretation, not.

Hence, in the short interpretation, if an array `A` is broken, then `A`, `A[0]`, `A[0][0]`... are broken, till one element `B` is reached which is not (which may be the singleton), thence no elements is broken: `B[0]`, `B[0][0]`, etc.

4 Relational operators and empty selections

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 op 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];
```

And there is one obvious possibility missing: comparing all the elements yielding a single value, assigned to `E` in the example. There is no need for a new operator, just syntax for selecting a 0-dimensional selection.

Therefore, the right choice is to treat `==` and `!=` as any other binary operator and operate on the selected elements. There is no need for the duplication of all comparison operators.

We define A `!=` B to be `!(`A `==` B`)`. Here, the result of A `==` B is a matrix with all its elements 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] )
```

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 us the succession of two tokens. C++ already has a similar problem with the `<<` operator, whereby nested templates need a dividing space between two consec-

utive `<`.

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, without any doubt the `A[]` syntax should be preferred over the `[A]` one.

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 of `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, but

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

are not, because `D` decays to a pointer.

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

All four calls are equivalent since the selection is lost (in our design) when the matrix is

passed as argument to function, but it seems to us that the first one is the more natural one.

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

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

Again, 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, and this is also the case when used for the operands of the `==` and `!=` operators, though here there is a parallel with matrices carrying "smaller" selections.

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 is only one such situation:

```
&A[]
```

(arbitrary arrays with selections will be excluded from other places, but arrays containing very simple selections will not). Arrays with selection will be precluded from these places because a different semantics may be wanted in the future for arrays with selected elements and specifically 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 there:

```
int A[5][5];
&A[]; &A[0][]; //Allowed
&A[0:1][]; //Not allowed
```

This agrees with our specification that a `[]` selection applied to an array which already carries a selection has no effect.

5 Restrictions on arrays with selection

Inconvertibility to pointer

The section on array subscripting suggested that an array with a selection should not be convertible to a pointer. Far from being a defect we see this as an advantage, for it opens the door to functions taking an array as an argument. it could be like this:

```
int sum_points(int points[:6]);
double determinant4(double A[:4][4]);

int main(void){
    int s, points[6];
    double x, A[4][4];
```

```
    x = determinant4(A[]);
    return sum_points(points[]);
}
```

Restrictions for broken arrays before lvalue conversion

The discussion on margins showed that it is better to exclude an array with selection in the long interpretation where the difference between one and the other choice (to keep margins or to drop them) would be made observable. It is also not a good idea to allow arrays with the first dimension broken where the semantics is different whether the array is trimmed or not (i.e., after and before lvalue conversion). In some situations it is only a broken first dimension that matters. 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])
Long interpretation
sizeof(B[:][0:5])        // = sizeof(B)
sizeof(B[2:3][0:5][0]) // sizeof(int[10])
Short interpretation
sizeof(B[:][0:5])        // sizeof(int[10][5])
sizeof(B[2:3][0:5][0]) // sizeof(int[5])
```

However, in the long interpretation **sizeof** would yield a different result after lvalue conversion, agreeing with the values of the short interpretation.

Long interpretation

The operand of **sizeof** does not undergo lvalue conversion (6.3.2.1), but the dichotomy seems us worth enough to exclude broken arrays from the **sizeof** operand:

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

Not all unbroken arrays should be allowed, though: there is a problem when either the length of the array or the expression L used for the selection is not an ICE. For example:

```
int A[2][n], B[2][5];
sizeof(A[:][0:3]);   sizeof(B[:][0:n]);   sizeof(B[0][0:n]);
```

Whether the three arguments to **sizeof** are unbroken or broken depends on the value of n. The problem does not arise if the selection is [:].

To avoid this, the range selections in the array should be restricted to [0:L] with L an

integer constant expression or `[:]`, and in the first case, if not the outermost selection, that the array on which the selection is operating be a fixed length array. But if `[0:L]` is not the outermost selection the user can and should simply omit the selection. Since allowing those selections would complicate the wording for **`sizeof`** for little gain if at all, we do not allow such not-outermost selections.

The text on the allowed arrays with selection as operands of **`sizeof`** finally would read (in case the long interpretation had been chosen):

If its operand is an lvalue and is an array carrying a selection, it shall be of the form `A[R]E` *where* `A` *is not an array with a nonempty selection,* `[R]` *is a range selection of a form other than* `[B:L:s]`, *and* *E* *is empty or a succession of range selections of the forms* `[]`, `[:]` *or* `[::]`.

Note that the selections `[]`, `[:]` or `[::]` which follow `[R]` can be ignored by the translator; they do not affect the result of the **`sizeof`** operator. Also, we had to take care not to exclude arrays with selection which have already undergone lvalue conversion.

This wording excludes a selection like `B[:][0:5][0]` with `B` as above, which is an array carrying a one-dimensional, unbroken selection. Allowing an array subscripting expression which results in an array with selection complicates the wording, again for little or no gain. The user should simply write in this case `B[0][0:5]`, which is always allowed as the operand to **`sizeof`**, whether `B[0]` has length 5 or more.

If the **`sizeof`** is part of the replacement text of a macro, that macro cannot control the way the argument is passed to it. For example, it cannot preclude `B[:][0:5][0]` in place of `B[0][0:5]`. But in macros even more that directly in the code, calling **`sizeof`** on its argument is more dangerous, because of the dependence on the lvalue conversion, for example

`sizeof``(A[:][0:3])` vs. **`sizeof`**`(+A[:][0:3])`

and because the array passed to the macro may or may not be broken.

If a writer of a macro wants to apply **`sizeof`** to its argument whatever it be, what is needed is a mechanism for making an array forget its selection. For example

`sizeof``(_Unselect(x))`

But it may be that it is the size after lvalue conversion what the macro wants. See the section “typeof, sizeof, _Unselect() and _Value()” in “Further extensions”.

Short interpretation

The value “returned” by the **`sizeof`** operator is the same before and after lvalue conversion. 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., **`_Lengthof`**`(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 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, and **`sizeof`** would not return the same value for all objects of type *T*`[`*l*`]`.

& operator

Long interpretation

In a first version we chose the resulting expression to be a pointer to the full array, the selection being forgotten. This is because the purpose of selecting elements is to perform a range operation, which is not the case if the **&** operator is applied to the array with selection:

```
int A[10], B[10][8];
&A[3:5]              // int (*)[5], points to A[3]
&B[:][0:2];          // int (*)[10][8]
&B[5:5][0:2];        // int (*)[5][8], points to B[5]
&B[5:5][0:2][0];     // int (*)[8], points to B[5]
```

We think it could be interesting to allow the addressof operator to construct a pointer that can address the selected multi-dimensional subarray. So as not to close this possibility for the future, we disallow the **&** operator.

Short interpretation

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 memory of the layout is copied with the value:

```
int B[10][8], C[6][4];
int (*p)[5][broken 2], (*q)[5][broken 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; 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. And 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 more than the present proposal intends. Therefore, we disallow the **&** operator.

typeof

Long interpretation

In the first place we restrict to operand to **typeof** in the same way as the operand to **sizeof** so as not to close the door for a future change in the meaning of **typeof** for broken arrays (which exist only before lvalue conversion) and to avoid a result that may depend on choices that may change (e.g., margins): `typeof(x[2:6:2][0:5]) y;`

In **typeof**, in addition, it may be decided in the future that its result remembers the selection. This is almost as making arrays with selection a different type than the full array. This is not the semantics of the long interpretation, where we want broken arrays to have the same type as the corresponding full array and let the semantics of the selection be taken care of by the concept of "carrying a selection". If it were remembered, it should then be ignored in contexts where it cannot apply, as in a declaration:

```
typeof(x[:][0:5]) y;
```

But what to use **typeof** for if not a declaration or the type of a compound literal? For a cast. A cast is applied to a value, not to an lvalue, so here arrays with selection that change their number of elements after lvalue conversion are even more inadequate in **typeof** than in, say, **sizeof**:

```
(typeof(A[0:n:2])) A[0:n:2]; //Different types
```

Furthermore, if the programmer wants to recover the selection, this can easily be done because the result of a cast is not an lvalue (hence, it consist of only the elements that had been selected), by applying `[:]` or `[::]`:

```
(typeof(+A[:][0:5]) A[:][0:5])[::]
f(n, ((typeof(y[0:n])) y[0:n])[:] )
```

So in the end we decided to allow the operand to the typeof operators in the same cases as it is allowed for the **sizeof** operator, and the result is the type of the expression with no selection.

Another reason for preventing for the time being the application of typeof to an array carrying a broken selection is that it may be to the type after lvalue conversion that it is more useful to apply the operand. That cannot be achieved by applying a trick for enforcing lvalue conversion:

```
#define typeofl(x) typeof(+(x))
typeofl(x[:][0:5]) y;
```

for it will also apply integer promotions and is only possible if the selected elements are singletons of arithmetic type.

Short interpretation

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. This will 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. For this reason we could allow **typeof** in the same cases as **sizeof** is allowed in the long interpretation; namely, 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**.

typeof_unqual is allowed on any array with selection, be it broken or not.

_Lengthof

Long interpretation

The restriction for the operand would be similar to that of **sizeof.** However, here it is only the outermost dimension which matters.

Furthermore, the question arises of what we want **_Lengthof** to yield: the total length or the number of selected elements. This alternative arises in the long interpretation; in the short interpretation the only possibility is the number of selected elements, for in that case it is also the total number of elements. This possibility of two results in the long interpretation is not the same situation as in **sizeof**. Here it is clear that we want the total size; the reason not to allow a broken operand in **sizeof** is that that size will change upon lvalue conversion and that it is not clear what *the user* wants. He may be calling **sizeof** passing a broken selection and expect the operand to give him the size of the selection, not counting unselected elements. If, for **_Lengthof**, we settle on the number of selected elements, no restriction would apply to its operand.

To return the number of selected elements is our preferred option. An lvalue is short lived, and we see no possible use of the length of it including the unselected elements. Furthermore, with this interpretation it is possible to use **_Lengthof** on an operand whose outermost dimension was some inner dimension but has been exposed to the outermost one as a result of array subscripting, without the result depending on whether margins are retained or not when array subscripting is applied. It is also the sensible choice for multidimensional indexed or direct selections, to be explored below.

Therefore, for this operator, we resolve to let it return, in the long interpretation, the number of selected elements of the outermost dimension (i.e., of elements of the array proper), whence no restriction applies to its operand.

Since we insist in making `A[]` always equivalent to `A`, **_Lengthof**`(A[])` should return the same as **_Lengthof**`(A)` and not 1, which seems and ad hoc exception. But it is not: **_Lengthof** returns the number of selected elements from the first dimension, or the number of all of them if there is no selection there. And indeed `A[]` carries no selection in the first dimension. If it were so, a further selection `A[][B:L]` would select from the second dimension, which it does not.

We may say that `A[]` carries a selection in its 0-th dimension, if we start counting dimensions from one. This dimension has a single element, which is the whole matrix. As with any element, its elements run along those of the next dimension. Thus, the elements of `A[0][0]`, which is an element form `A`'s second dimension, vary along `A`'s third dimension; those of `A[0]` run along `A`'s second dimension, and those of `A[]`, which are those of `A`, run along `A`'s first dimension.

But we would place a note in the wording pointing out that an array carrying an empty selection carries no selection in its first dimension, had we chosen the long interpretation.

Short interpretation

Here the number of selected elements equals the total number of elements, and consequently there is no doubt as to what **_Lengthof** should yield.

Other Restrictions

An array with selection should not be allowed as the argument passed to a function: it cannot and should not decay to a pointer. 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 it does not act *as a range of objects* in our terminology, its use is questionable. We have identified these places:

Argument to function

_Generic, controlling expression

auto, initializer for a type inferred declaration

alignof

[*k*] operator

unary *

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

_Generic

Since it is the type of the controlling expression which is needed, there seems to be no reason for selecting elements from the array. This by itself is no reason for forbidding its use here, but we would do so so as not to make the code depend on broken arrays being of the same type as the corresponding full array, as is the reason for forbidding them in **typeof** in the long interpretation.

But the controlling type of a **_Generic** selection is taken from its controlling expression after lvalue conversion, so this wariness does not apply. Indeed, 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 any change in the wording for this operand. Arrays with selection will be allowed there. In the long interpretation their unselected elements are lost after lvalue conversion; in both interpretation brokenness is lost; in any interpretation the array type is preserved.

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.

[*k*] operator

The `[k]` operator presents the ambiguity in the long interpretation of whether to start counting from the 0-th element of the array or form the first selected one. We have already argued that the latter choice is the right one.

unary *

This is impossible, since the array with selection should not be convertible to a pointer.

Macros, selection forgetting and lvalue conversion

Long interpretation

Many of the cases that needed careful consideration above will not arise in practice. Why would a programmer write

```
typeof(+A[6:3][0:5]) B;
```

instead of

```
typeof(A[0][0]) B[3][5];
```

? The latter has the advantage that integer promotions are avoided. Similarly, if the user wants the size of `A[6:3][0:5]` after lvalue conversion, he can just write

```
3*5*sizeof(A[0][0])
```

And if he wants the size before lvalue conversion he should write `sizeof(A[6:3])` or `3*sizeof(A[0])`.

Both interpretations

For plain arrays, **typeof, sizeof** and **_Lengthof** are useful because the array may be declared in one place and the operand used in another place where the declaration is not visible (in the literal sense). An array with selection is used on the spot. Why would someone write

```
_Lengthof(A[0:n:3])
```

? The value is either `n`.

These situations arise only in macro definitions. For example, to write a macro that selects the first three elements of an array:

```
#define invert3(x) (x)[0:3] = 1/(x)[0:3]
```

If the array `x` passed to the macro already carries a selection of singletons (e.g.: `float x[10]; invert3(x[5:5])`), the macro will fail. Instead of providing in the language some operand to forget the selection, the best solution is probably to write the macro as above and do not pass to it an array with selection in its innermost dimension:

```
float *p, x[10][6];
invert3(p);          // O.K.
invert3(x[:]);       // O.K.
invert3(x[:][:]);    // Wrong
```

Likewise, if a macro wants to set to zero its argument `x`:

```
#define setzero(x) memset(&(x),sizeof(x),0)
```

it will fail if `x` is an array carrying a not-outermost selection of type `[B:L]` or a selection `[B:L:s]` anywhere. Far from being a defect, this is a safe behaviour: What would the caller want in those cases? To set to zero the whole matrix or just the selected elements? If he wants the whole matrix there is no reason for passing the matrix with a selection to the macro, except for a selection in the first dimension in case he wants to cut down the matrix; but in this case the macro does work: `setzero(A[0:n])`. If just the selected elements

are wanted, the way to do it is using the range operations!

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

If the macro knows its argument will be a matrix, using range operations is also the right way to define it:

```
#define setzero(x) (x)[::]=0
```

This will set to zero only the selected elements of `x`, if they are singletons, the singletons from `x`'s selected elements if they are arrays, or the whole matrix if there is no selection.

Therefore, the cases where a macro cannot attain complete generality with respect to the type and selection carried by its argument are in the first place very rare (a multidimensional selection need be present, or a user passing an array with a selection that the macro wants discarded, in which case the users should have simply passed the array with no selection), and the failure of the macro appears as a safety valve rather than a limitation of the same.

For this reason there does not appear to be a need for the introduction of one operator for forgetting the selection and another one for forcing lvalue conversion (avoiding, e.g., integer promotions). We explore them in "Further extensions".

6 Casts

Restrictions

Casts apply to values, not to lvalues. This simplifies the design. In the long interpretation, we'd like to restrict the cast operand in the same way as the **`sizeof`** operand, and since arrays with selection which are not lvalues are allowed there without restriction, there is no restriction in the operand to a cast. Also, there is no need to leave the possibility open for a pointer type in the cast that would result in a pointer addressing only the selected elements, as we did for the `&` operator, since after lvalue conversion all elements are selected, and even more fundamentally, a value of array type cannot be cast to a pointer to its address because it has no address.

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 which do not decay, these should be allowed to be cast to arrays.

Changing the singleton type (I)

The only obvious type a value of array type can be cast to is its own type. A cast changing the singleton type would also be meaningful:

```
double A[4][4];
(float[4][4]) A[::]
```

This can be used to prevent warnings from the compiler:

```
float A[4][4], B[4][4], C[4][4];
```

```
A[::] = (float[4][4])(B[::]+C[::])[::];
```

The last `[::]` is there because we want the cast to forget the selection. The reason is the the cast is to an "array of four arrays of four float", and there is no selection in that expression; i.e., in the type written within `()`. Also, this choice adds flexibility, since the user can apply any selection he wants to the result of the cast. In particular, the selection before the cast can be recovered by writing `[::]` or a series of `[:]`.

At first we didn't include these casts in the present proposal, because it seems it requires the compiler to make a copy of the object, transforming each of its singletons. Later on we changed our design. This will be explained soon below.

Redimensioning cast

An array may be cast to an 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) _Lengthof(x)*_Lengthof((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[];
(int[2][3][6])A[];
```

Changing the singleton type (II)

Coming back to casts that change the singleton type, consider

```
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:

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

for example.

We therefore changed radically our original design for casts. By making it a range operation it gains in expressivity, just as the `==` operator (see below). 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. In the former case we have the redimensioning cast of our original design, that now requires the array to which it applies to carry the `[]` selection, no other one is valid. In the latter, the type of the cast can be anyone that is allowed for the singleton. We refer to these two casts as the *array cast* and the *range cast*. If intermediate selections were allowed these would be of array type, so that the compiler need not create a copy of the object and the effect would be to just reinterpret its dimensions.

If `B` is of type `double[4][4]`, the type of an expression like

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

is `float[4][4]`. What then, is the advantage from the point of view of the translator with respect to our first design of `(float[4][4])B[::]`? The advantage is that in this latter case the selection is forgotten (as we argued above), which result in a plain array of kind `[4][4]` for which the compiler may need to reserve space in memory; for instance, selections can be applied to that array. In the cast `(float)B[::]` the result is an array carrying 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. Hence, no copy for it in memory is needed.

Variants of the element type in the cast

In an array cast we allow casts to an array type with a compatible type for the singletons. This is more restricted than a compatible array type, which includes variable size arrays of any size (the requirement for two array types to be compatible only mandates the sizes to be the same if both are given by integer constant expressions). 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.

The selection after the cast

Our original design where the cast forgets the selection is flawed in one respect: a value of array type without selection cannot exist. If it did, it would have to decay to a pointer in many situations, which is impossible since a value hasn't got an address. Therefore the result of an array cast must be a matrix which cannot decay to a pointer; i.e., a matrix carrying an empty selection. In the range cast we finally designed, the result of the cast carries a selection of singletons, and in the array cast the result carries an empty selection, as we have already explained.

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 may be an array matching the dimensions of those selected elements and with a 0-dimensional selection, to prevent it decaying to pointer, as here. We do not allow such a right operand to carry a (>0)-dimensional selection:

```
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 or forgetting the selection, if it were possible. The latter would require selecting again with `[]` to prevent the matrix from decaying to pointer:

```
int D[4];
D[:] = A[0][:]+B[:];
C[:][:] = D;
C[:][:] = _Unselect(A[0][:]+B[:])[]; //Supposing it existed
```

The right operand may also be an array whose selected elements match the left operand's selected elements:

```
int A[3][4][5], D[3][5];
A[:][:] = D[:];
```

That this is to be allowed follows from the general rule that `A[:]`... *op* `B[:]`... means `A[`*i*`]`... *op* `B[`*i*`]`..., 0≤ i <`_Lengthof(A)`. Hence, in this example, `A[0][:]=D[0]`, etc., and we have just explained that allow this (more formally, `A[0][:]=D[0][]`; i.e., `D[0]` does not decay to a pointer). For the same reason, the following is allowed:

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

```
A[::] = B[::]; // Equivalent to this
```

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 (the 1st case), as the not allowed `A[:] = B[:]` above, or whether it is a per-element assignment as the latest `A[:] = B[:]` above, (the 2nd case). 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.

The allowed `A[:] = B[]` needs the `[]` after `B` just to prevent it decaying to a pointer. We cannot just say in a situation like this that `B` without a following `[]` does not decay to a pointer, because we may want it to decay, if the type of `B[0]` is compatible with the element type of `A` (i.e., fill all elements of `A` with the address of `B[0]`):

```
int *A[5], B[4][8];
A[:] = B[1];   // A[i]= &B[1]
```

Assigning into an array

We considered allowing the assignments in the right columns as synonyms of those at the left:

```
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][];
```

But we want range operations always to involve matrices with selections or singletons: if one operand carries a selection then the other operand either carries a nonempty selection or is operated with each of the elements selected from the first operand. Therefore, of the forms at the right column we only allow the last one (where the “first operand” is `A[0][]`, with just one selected element, namely `A[0]`, and the “other operand” is `D`).

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 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 not 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 can 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.

8 Other

The decaying of 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 **6.3.2.1** on arithmetic conversion says that

> *Except when it is the operand of* [...] *an expression that has type "array of* type*" is converted to an expression with type "pointer to* type*" that 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 description could be preserved with extra wording, either specifying a conversion of the array with selection to a pointer, which goes against its intended use, or preventing it to apply to arrays with selection and inserting a description for the later directly in terms of "selects the *n*-th element", which would leave a strange asymmetry: "why not apply the direct description also to arrays which have no selection?", would think a reader of that text.

The redefinition of the `[`*k*`]` operator is needed for other, unrelated reasons. Because of this, this redefinition was moved to an independent proposal and the proposed wording here assumes that to have been already integrated into the standard.

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 an ICE with the 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 to `0`. But in the example above `A` is declared as an array of integers, so the values `0` it contains cannot be pointers. If the programmer wants those kinds of combinations both arrays shall be declared with elements of

pointer type. E. g.,

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

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

We do not allow the following:

```
int *A[3], B[3];
A[:] - B;    //Would be {A[0]-B, A[1]-B, A[2]-B}
```

This affects the operators **-**, **<**, **>**, **<=**, **>=**, **=** and **-=**. If the programmer wants this, the code should be written thus:

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

It seems to us that allowing the mixing would be a source of bugs. The workaround, as shown in the examples above, is straightforward.

Arrays with selections of different depth

We allow them as in the third assignment below:

```
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 selec-

tions of singletons. In the third one, two matrices carrying selections of singletons but not matching.

When both operands are arrays carrying a nonempty selection the general rule dictates that each of the corresponding selected elements are 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, a similar rule says that each selected element is operated with the singleton. When applied to the expression `A[::] += B[::]`, which is equivalent to `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 `A[`i`][`j`]`. When the rule is 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. either `a` or `b` still carries a nonempty selection and the other one is a matrix carrying no selection or an empty one.

The first case will be the most common one and needs no wording. The second possibility is generally not allowed, and no wording will be provided for it in general. In the third case, a rule will be stated according to which the innermost selected elements of the array that still carry a selection shall match the dimensions of the other operand, and each selected element will be operated with that other operand, singleton per singleton (thus, if `a` represents each selected element and `B` the operator which no longer carries selection, `a[::]` *op* `B[::]` is performed).

Thus, the cases 1 and 3 are treated in the same way: each selected element from the arrary which still carries a selection is operated with the other operand, and one and the other must have matching dimensions.

Assignment, equality and relational operators

With respect the third case, assignment operators 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 and is right as stated.

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. As respect to the third case, the result is not as defined by the general rule, but a single element for each `a` *op* `B`, as we have seen.

For relational operator the third case is forbidden.

Multiplication of a matrix by columns

The rule forthe 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
```

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 with selection, which can be the left operand of an assignment. 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 the singletons be modifiable lvalues.

This choice also accommodates better arrays at the left of an assignment as in `E = B[]`, where `E` does not carry a selection. Otherwise, in order to allow the latter construction, the wording on the assignment operators has to be adjusted to include specifically arrays as possible left operands beyond modifiable lvalues, or to accept that they are allowed by saying that `E=` is equivalent to `E[]=`. Contrary to arrays decaying to pointer for, e.g., the addition operator, here the selection `E[]` does not transform the array into anything different. In short, it would be wrong to pretend that arrays without selection continue to be not modifiable. We do still say that `E=` is equivalent to `E[]=`, but this is just to be able to describe those assignments by the same words used for assignments where the left operand carries a selection.

Graded complexity of array selections

The range selections proposed in this paper are graded in complexity of semantics and interaction with the current language. The following is an informal list of growing complexity or of dependency, later features requiring prior ones. In many cases there is not a growth in the complexity of implementation. Here, `A` does not include itself range selections, i. e. is an array without selection; `[`k`]` represents the array indexing operator; whenever `[B:L]` is present it is understood that `[:]` or `[B:L:s]` is also possible, and ICE stands for integer constant expression. The items in grey are meaningful for the long interpretaion, which we finally did not take:

- `A[B:L]` with `B`, `L` and `s` ICE . `[`k`]` is allowed.
- `A[B:L]` with `L` and `s` ICE. `[`k`]` is allowed.
- `A[B:L][B':L']`... , `L`, `s`, L', `s`' ... ICE. `[`k`]` not allowed on multidimensional selections.
- `A[B:L][B':L']`... , `L`, `s`, L', `s`' ... ICE. `[`k`]` allowed when the second dimension is not broken (This dimension will become the outermost dimension of the selected array.)
- `A[B:L][B':L']`... , `L`, `s`, L', `s`' ... ICE., with `[`k`]` allowed anywhere.
- `A[B:L][B':L']`... , `L` and `s` might not be ICE but only in inner ranges, not in the outermost dimension. `[`k`]` allowed when, in the the second dimension (which will become the outermost one of the selected array), `L` is an ICE.
- `A[B:L][B':L']`... , `L` and `s` need not be ICE. `[`k`]` allowed anywhere.
- `A[B:L][B':L']`, where `A[B:L]` is an array of pointers.

Independent from these, the following restrictions on the value of `s` are also graded in difficulty of handling.

- `s` might not be zero.
- `s` might be zero only if `L` is an ICE equal to 1.
- `s` might be zero only if `L` is 1.
- `s` might be zero.

With respect to the broken character of selections, their length and the type of the selected elements, the following is an increasing list of allowed selections:

- `A[R]` where `A` is a unidimensional array not carrying a selection, `R` can have any form except empty and `B:L:s` and if it has the form `B:L`, `L` is an ICE.
- `A[::]` where `A` need not be unidimensional and if it carries a selection it must be of the above form.
- Like the above, with spurious `[]` and `[::]` allowed.
- `[]` and `[:]`
- `[B:L:s]` allowed in the first selection, with `s` an ICE of value 1 or -1.
- In addition, `s` need not be an ICE.

- `L` need not be an ICE.

------ *Up to here the arrays are not broken* ----

- `s` an ICE, but need not be `1` or `-1`.
- Any other combination for the first selection
- Any other combination

This proposal includes all possibilities listed in the foregoing lists except the last one of the first list.

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 in the implementations, which requires standardising common extensions and, when possible, foreseeing it. 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 too make range selections mandatory:

`__STDC_ARRAY_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 reason we name it array_selections and nor range_selections is that other array selections are possible, though we do not provide wording for them (indexed and direct selections).

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

`__STDC_N0_ARRAY_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 version. 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" with respect to direct selections below).

The different gradings above provide a hint as to the partial implementations of range selections that implementations may choose. Implementations defining **`__STDC_ARRAY_SELECTIONS__`** to one may be allowed to accept a partial set of possible combinations for the range. 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.

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

`__STDC_ARRSEL_CONSTANT__` Expands to `1` 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 `0` otherwise.

Here the values of `0` and `1` are reversed: The value `0` means more cases supported, the value of `1`, not supported. The reason for this is that we are not including this macro in the present proposal. If later on it is introduced, a value `0` should mean the same as its absence.

If the first 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 selections 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.

The reason why an ICE expression is not enforced for `B` in the outermost selection when `__STDC_ARRSEL_CONSTANT__` is `1` is that a variable base can be achieved even with `B` restricted to a literal `0`, as in `(A+n)[0:L]`. This is equivalent to `A[n:L]`, so mandating an ICE in place of `B` seems pointless. The reason why `B` is required to be constant in inner selections is that the restriction expressed by this macro means that all selected elements are at a translation time known offset from the base of the array (if not applied to an array with a VLA element type).

A value of `0` for the first 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];
```

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 none / partial in a specific form / full support, 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 allowing the restriction (of not allowing a step zero) only in case the implementation supports only constant `L` and s, i.e., in case the macro `__STDC_ARRSEL_CONSTANT__` were `0` (by then we were including this macro in the proposal), 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.

The macro `__STDC_ARRSEL_CONSTANT__` becomes more significant when *direct selections* are included, one of the topics of the next section.

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 selections 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 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. Yet we can 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 operands

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

in addition to those unary operators that produce a number (**`sizeof`**, etc., `[]` subscripting) and **`_Generic`**. Those four above are, not by accident, the ones that introduce sequence points. A recent proposal to introduce more sequence points and impose an evaluation order therefrom should take into account its impact for the translation of range operations.

10 Indexed and direct selections

An array with selection as the index

As in the following example:

```
size_t n, I[3];
float A[10];

I[] = ((size_t[3]){0,n,n+1})[];
A[I[:]]; //{A[0], A[n], A[n+1]}
```

The reason for requiring the selection `[:]` after `I` is apparent when considering multi-dimensional arrays. Suppose we don't require it and take the elements of the array placed inside `[]` as the indices (the *first interpretation*, which will be referred to later on). Suppose we don't require it:

```
size_t I[4][2];
float A[8][8];
I = (typeof(I)){{0,3}, {2,2}, {4,1}, {6,0}};
A[I];          // {A[0][3], A[2][2], etc.}
A[I[0]];       // This should be A[0][3]
A[{0, 3}]      // But is {A[0], A[3]}
```

Our first choice to provide all options and a coherent selection scheme was that a one-dimensional array specifies a sequence of indices to be applied to successive dimensions. Therefore, placing `I` as defined above inside an array selector `[]` has no meaning. If we

want a series of elements to be selected we need to provide a range of arrays:

```
A[I[:]];      // {A[0][3], A[2][2], A[4][1], A[6][0]}
A[I[0:2]];    // {A[0][3], A[2][2]}

size_t J[2] = {0,2};
size_t K[2][1] = {{0},{2}};
A[J[]];       // A[0][2]
A[J[:]];      // {A[0], A[2]}
A[K[:]];      // {A[0], A[2]}
```

The last line has the displayed meaning because of the way we have defined `[(size_t[]) {0}]`. The next to last line because of the way `[0]` is already defined.

The rule is: The selection `A[I[R]]` has as many elements as elements are selected in `I[R]`.

This criterion also has the advantage that array subscripting operations, where the expression `n` in `[n]` has integer type, and array selections with an array as the index —which we will call *indexed* selections—, can be distinguished visually. The latter will always feature a selection. The only exception is the selection of only one element, as in

```
size_t I[3] = {2,1,1};
int A[4][4][4];
A[I];         // A[2][1][1], since a nondecaying I is equiv. to I[]
A[I[]];       // The same as above
A[I[0]][I[1]][I[2]];
```

The three selections are equivalent (later it is seen that they should not be equivalent; see the section below “Singleton or not”). We believe it is the third which should be used, not the first (that is, when we were considering this design). Those three options are always available to the programmer.

The kinds of matrices allowed as indices

The index matrix shall be one- or two-dimensional. More specifically, `I[0]` should be:

— of integer type; or

— A fixed length array whose element type is an integer type.

Furthermore, if the matrix is bidimensional it may not carry a depth-two selection.

Two- and higher-depth selections could be possible provided the type of the selected elements is one of the above, but it seems to us an unnecessary complication.

Apart from the type of the matrix we may restrict the form of the expression. We may forbid computed arrays, as in `A[I[0:n]+J[:]/2]`. These computed arrays complicate the translation. If the array expression is given by an identifier of array type, to which possibly an array selection has been applied, the indices of the elements chosen in `A` are taken directly from an array, say `I`, `A[`i`]` needing `I[`i`]`. For example, the following instruction could be translated as shown:

```
A[I[:]] = B[:]+C[:];   //Assume the length is n
for i=1..n  A[I[i]]=B[i]+C[i]
```

But if the index matrix is computed this is not possible. However, a compiler able to trans-

late a range operation can place inside `A[]` the operation that yields the *i*-th element of the index matrix, avoiding the need to allocate memory for the computed index matrix:

```
for i=1..n  A[I[i]+J[i]/2]=B[i]+C[i]
```

It seems therefore that the complication is more apparent than real. We prefer that not restriction appies to the index matrix in this respect. But just in case it be desired to apply, in a first version, a restriction on the index matrix, here follows a proposal on how to formulate it:

Our first idea was to require that the first token after the opening `[` be an identifier. This would make invalid a valid expression by enclosing it in `()`. There is no case in the language where this happens for an expression. We may modify the requirement to say that "after all outermost pairs of matching `()` have been removed the first token is an identifier". But this would leave out expressions like `(I[0])[:]`. We may strip all pairs of redundant parentheses, but that would still miss strings and compound literals. In the end we came to a wording which describes the possibilities for the expression after removal of outermost `()`, leading to a recursive specification, which is equivalent to saying that the expression must match *direct-array*, defined as follows:

array-kernel:	*direct-array:*
`(` *direct-array* `)`	*array-kernel*
constant (of array type)	*array-kernel* `[` *cond.-expr.* (of integ. type) `]`
compound literal (of arr. type)	*array-kernel range-selector*

avoiding the definition of these two terms in the syntax.

Then we realised that a semantic definition is much easier. What we want in the end is that the indices be retrieved from some array which already exists in memory. Thus, we may say that the index expression has to denote and object, or even better, that it has to be an lvalue.

Limit to nested indices

We may allow implementations to impose a limit on the nesting of indices in array selectors, listed on 5.3.5.2 Translation limits:

— 15 nesting levels of index arrays in array selectors

We were guided for this by the "12 pointer, array, and function declarators (in any combinations)" and the "63 levels of nested structure or union definitions". We think that the latter is the one that best matches the case at hand, but at the same time see index nesting in selectors much less likely than nesting of structures.

An array without selection, sometimes

While we argued that mandating the index to carry a selection facilitates visual discrimination of array subscripting and range selection, having to always type `I[:]` may be felt as a spurious nuisance. If `I` is two-dimensional there is no other possible interpretation for `[I]` (since we are precluding `I[:][:]`); also if the array to which the selection is being applied is one-dimensional, which will probably be the most common use. The programmer may tell apart easily a range selection of this kind from array subscripting by, e.g., using always a capital letter for the index matrix and a lowercase one for scalars, or by using only `A`,`B`,`I`,`J` for matrices that may function as indices, say.

Therefore, we propose that a selection in the index matrix be mandatory only if the

matrix to which the indexed selection is applied is >1-dimensional and the index matrix is one-dimensional. Thus,

```
size_t I[]={2,1},  J[][1]={{2},{1}},
       K[1][2]={{2,1}},  L={1,3,0};
int A[6], B[6][6], C[6][6][6];
A[I], A[J];
B[J];               // {B[2], B[1]}
B[K];               // B[2][1]
B[I[]],  C[I[]];    // B[2][1], C[2][1]
B[I[:]], C[I[:]];   // {B[2], B[1]}, {C[2], C[1]}
C[L[]];             // C[1][3][0]
C[L[:]];            // {C[1], C[3], C[0]}
B[L[0:2]];          // {B[1], B[3]}
B[L[:]];            // {B[1], B[3], B[0]}
```

An implementation might relax this further to require a selection only if the number of elements of the matrix used as the index is ≤ the number of dimensions of the matrix at the left, so that in the last of the examples above `[:]` would not be needed: `B[L]`.

We have to be careful not to break the equivalence between `I` and `I[]`. Since, when `A` and `I` are one-dimensional, we have made `A[I]` equivalent to `A[I[:]]`, so shall be the meaning of `A[I[]]`. We cannot say that in this case the selection of `I` is ignored because it is still needed if the programmer does not want all the elements in the index array to be selected:

```
float A[10];
int I[3]={0,9,3,6};
A[I[1:3]];
```

So we simply state that "if the elements of `A` are singletons the selection `A[I]` is equivalent to `A[I[:]]`". Likewise if neither the elements of `A` nor those of `I` are singletons. So the rule reads

if the elements of `A` *are singletons or the elements of* `I` *are not singletons the selection* `A[I]` *is equivalent to* `A[I[:]]`

An array without selection, always

The previous rule leaves as exceptional the case when the elements of `A` are not singletons and the elements of `I` are. We prefer not to have this exception. This means that if the user wants to apply `{2,1}` as one-element selection of depth two, he will need an extra `{}` surrounding; i.e., a bidimensional matrix as index:

```
int A[6][6], I[]={2,1}, J[1][2]={{2,1}};
A[I];    // Equiv. to A[I[]] and A[I[:]]. {A[2], A[1]}
A[J];    // Equiv. to A[J[]] and A[J[:]]. A[2][1]
```

The selection `A[I]` *is equivalent to* `A[I[:]]`

We have come round to the first interpretation, which we deemed wrong, but now we don't think `A[I[0]]` should be `A[0][3]`:

```
I = (typeof(I)){{0,3}, {2,2}, {4,1}, {6,0}};
A[I];          // {A[0][3], A[2][2], etc.}
A[I[0]];       // {A[0], A[3]}
A[I[0:1]];     // Now this is needed to get {A[0][3]}
```

(We are supposing that `I` has been declared before, with type T`[4][2]`, for some T.)

That the last two lines yield different results is not incoherent. `I[0]` is a 1-dimensional matrix while `I[0:1]` is a 2-dimensional matrix with only one element in its outermost dimension. Their types are T`[2]` and T`[1][2]` respectively. Similarly,

```
J = (typeof(J)){0, 3, 1, 2};
A[J];          // {A[0], A[3], A[1], A[2]}
A[J[0]];       // Array subscripting. A[0]
A[J[0:1]];     // {A[0]}
```

for `J[0]` is of type T and `J[0:1]` is of type T`[1]`.

The relation between the number of elements of the index matrix and the number of elements of the resulting selection is now that they are equal. When we mandated a selection to be always present in the index matrix, the number of elements of the resulting selection equalled the number of elements selected in the index matrix. Now this is still the case for nonempty selections, but is different for an empty one. Now we have

`_Lengthof(A[I]) = _Lengthof(A[I[]]) = _Lengthof(A[I[:]]) =`

`= _Lengthof(I) = _Lengthof(I[]) = _Lengthof(I[:])`

And generally, `_Lengthof(A[I[R]]) = _Lengthof(I[R])`

`R` here may be empty, `:`, `B:L`, `B:L:s` or an indexed selection itself. It may be `::` only if `I` is one-dimensional.

The relation between the depth of the resulting selection in `A` and the index matrix is that the former is the number of elements of the elements of the latter; i.e., of `R[0]` in our notation. In the selection `A[I[0]]` the index matrix is `I[0]`, and `I[0][0]` are singletons. In `A[I[0:1]]` the index matrix is `I`, and its single element is a bidimensional array: `I[0:1][0]`, which is `I[0]`.

The type of the selection

The type of a selection of the form `A[R]` where `R` is a matrix is obvious after lvalue conversion. If the elements of `R` are arrays of length m, or singletons in which case we let m=1, and they are l in number (the selected ones. In any case, `R` undergoes lvalue conversion), the selection has depth m and consumes m dimensions from `A`, and the type is

`typeof(A[0]`$\ldots$`[0])[`l`]`

where there are m `[0]`'s. Equivalently, if the type of `A`'s singletons is T and `A` has n dimensions,

$$T[l][l^{(m+1)}]\ldots[l^{(n)}]$$

Long interpretation

The type before lvalue conversion is not that clear. The values of multidimensional selections in `R` (the length m arrays above) can be mixed and repeated in any way. One may

choose to impose immediate lvalue conversion of an indexed selection; that is, an indexed selection expression is not an lvalue. This is not a good choice because it precludes one good use of indexed selections:

```
float A[10], B[4];
int I[3]={0,9,3,6};
A[I]=B[:];
```

Before lvalue conversion, the selection expressed by `A[R]` is contained in the object referred to by `A` and there may be holes. This is just as for range selections. Repetitions are part of the selection, but the repeated elements do not appear twice in the object, as is the case for a stepped selection of step zero. There is no other choice, since if it is still an lvalue we have to keep it inside `A`. The first m dimensions (continuing with the notation above) are broken in an irregular way.

We can make the resulting array with selection less complicated if we collapse the first m dimensions. Continuing with the notations above, the types of `A` and of the indexed selection are respectively:

$$T[l^{(1)}]\dots[l^{(m)}][l^{(m+1)}]\dots[l^{(n)}]$$
$$T[l^{(1)}\times l^{(2)}\times\cdots\times l^{(m)}][l^{(m+1)}]\dots[l^{(n)}]$$

In this way there is only one dimension with an irregular selection. Furthermore, with this choice the matrix does not change the number of dimensions upon lvalue conversion.

Example:

```
float A[2][3][4][5][6];
int I[][3]={{0,1,3}, {1,2,0}, {1,2,3}, {0,1,1}};
A[I]; // A matrix of type float[24][5][6] with selection [{7,20,23,5}[:]]
```

That is, if we use **A** to represent the object at `A` accessed with type `float[24][5][6]`, the selection is `{A[7], A[20], A[23], A[5]}`.

In all the above, if `A` carries a selection of depth k, instead of the type of `A` the type of `A[0]`...`[0]` has to be taken, where there are k `[0]`'s.

With this choice `_Lengthof` gives the number of elements selected in the matrix.

Short interpretation

In this case, the type before lvalue conversion is always the same as after lvalue conversion, except for the brokenness qualifier. Taking the same example as above, the type of `A[I]` is `float[4][5][6]`.

As left operand in an assignment

An array with an indexed selection can be used as the left operand of an assignment provided the selection has no duplicate elements. This embraces a stepped selection with a step equal to zero as a particular case.

Singleton or not

If the matrix `A` has n dimensions and the depth of the selection is m, the selection has n-m+1 dimensions. If therefore m equals n, the selection is a 1-dimensional array:

```
int A[5][6], I[][2]={{0,1}, {1,2}};
A[I];    // {A[0][1], A[1,2]}
```

If in addition the index matrix has one element we get a one-dimensional array with one element:

```
int J[][2]={{0,1}};
A[J];         // {A[0][1]}
A[I[0:1]];    // {A[0][1]}
```

Since obviously *n* is the maximum possible value for *m*, there is no way to obtain a singleton from an indexed selection, even after lvalue conversion.

Indeed, there is no way to obtain a singleton from any selection whatsoever. Array subscripting is needed.

Margins

Long interpretation

We have not yet fully specified the type of a selection like `A[I]` above before lvalue conversion. We have said that it is a matrix of type **`float`**`[24][5][6]` with selection `7,20,23,5`, but not what the resulting type is. It can be **`float`**`[24][5][6]`, but it can also be **`float`**`[19][5][6]`, where the number 19 comes from 23-(5-1), and `A[I]` goes from **`A`**`[5]` to **`A`**`[23]`.

if we have chosen stepped selections, no matter how complicated they be, to be restricted to the interval of selected elements, it seems we should do the same for indexed selections. To exhibit a plausible example,

```
float A[6][10];
int I[][2]={{a,b}₁, {a,b}₂, ... {a,b}ₗ};
A[I]; // Has type float[max{aᵢ*10 + bᵢ} - min{aᵢ*10 + bᵢ} + 1],  1 ≤ i ≤ l.
```

and has no margins.

Direct selection

When using indexed selections with a fixed number of elements in the selection, having to express it through an intermediate index matrix seems an unnecessary roundabout. Instead of having to write

```
int I[3]={0,2,n}; A[I];
or A[(int[3]){0,2,n}];
```

simply write `A[{0,2,n}[:]]` or, more simply, `A[0,2,n]`. We do not propose the former to be allowed, just the latter. This latter syntax production clashes with array subscripting, because the subscript in this construct can be “comma expression”, not just an assignment expression (which itself seems too much). We ignore this for the moment and consider it at the end of this section.

The interpretation of the list is that if its elements are singletons they represent the elements selected from the outermost dimension, while if they are themselves brace-enclosed lists then each of these lists represents a multidimensional selection, and obviously all the lists must have the same number of elements:

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

A selection like `A[{0,2}]` is not ambiguous because it cannot mean `{A[0], A[2]}`, which needs `[0,2]`. Nor it can be `A[0][2]`, for reasons already explained.

A direct selection is the same as a selection with an array as the index, composed of the elements of the given list. So, just as with indexed selections, the number of elements within `[]` is the number of elements of the selection.

The values of the `__STDC_ARRSEL_` macros should have the same meaning for braced selections than for indexed selections given by a compound literal.

Side effects in the expressions composing the list do not seem a good idea. Restricting them further to integer constant expressions or identifiers seems too much, since one may write `A[0,1,2, n,n+1,n+2]`, for example. We do impose (that is, if we did provide wording) that their values have to be nonnegative. As to the side effects, one argument in favour of allowing them is that, since

```
int I[3]={0,2,n++}; A[I];
```

is permitted, so should `A[(int[3]){0,2,n++}]`.

Constant range expressions

The recently introduced *constant range expression* (CRE) is another natural way of specifying a selection, and it can be combined with a comma-separated list of integers:

```
A[0, 3...10, 13]
```

Constant range expressions duplicate the functionality of range selections of the form `B:L` with `B` and `L` integer constant expressions. This is a reason strong enough to exclude them *if they were not already in the language*. Since they are, the obvious choice is to exclude range selections of the kind `B:L`, extending range expressions to include non-constant ones. But we have already argued that the specification *beginning* : *end* (or *beginning* ... *end,* the argument is not about the syntax) is a bad choice for C. The type of the resulting expression (after lvalue conversion at any rate) depends on the value of the length *l*, and in particular it should be given by an integer constant expression when it is a compile-time constant, as in `[2*n:4]`, which in the *beginning* : *end* form would become `[2*n:2*n+4]`. This explicit presence of the constant length is desirable not only in form but even more in substance for the generated program, to make it possible for the compiler to translate the expressions of which they are part into vectorial instructions.

Constant or not

CRE are, as their name implies, given by integer constant expressions. We like to keep it this way for array selection from a matrix. For any selection which is not constant the present proposal gives the programmer plenty of tools for expressing it; CRE are just another syntax for selections `[B:L]` with constant `B` and `L`. On the other side, we think the comma-separated integers need not be constant, for if `A[(int[2]){n,n+2}[:]]` is allowed it seems difficult to justify that `A[n,n+2]` is not.

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'', ... ,K(l)]`, `[{k(1), ... ,k(m)}', {k(1), ... ,k(m)}'', ... ]`

Here `B`, `L`, `s`, `k` stand for expressions of integer type, `b`, `e` for integer constant expressions, `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 `*`, *b*`:`*l*, *b*`:`*l*`:`*s* or *irregular*, where `*` means "all elements are selected" and an irregular selection is given by the index array or direct selection used for it, in which case the length of the dimension it applies to is the product of the lengths of the dimensions involved in the array.

The feature test macros for indexed and direct selections

`__STDC_ARRAY_SELECTIONS__`: The words "range selections" which refer to what is supported, have to be replaced by "array selections beyond empty selections".

`__STDC_ARRSEL_STEPPED__`: If indexed selections are allowed, stepped selections have to be allowed too, including those with a step equal to zero, for it is just a particular case of indexed selection; i.e., a stepped selection can be achieved by means of an indexed selection, though if the length of the former is not an integer constant expression a variable length array may be needed for the index matrix.

Therefore, there are two possibilities for this macro with regard to indexed selections: That a value `1` implies that both stepped and indexed selections are supported, or that a value `1` implies only support for stepped selections and support for indexed selections is communicated by a value of `2` here. We prefer the second one, which splits support for indexed selections from support for range selections.

Direct selections require the translator to place the indices in the translated program. this is different from indexed selections, where the compiler need only replace A`[`*i*`]` by A`[I[`*i*`]]`, where A and I are the addresses of the array denoted by the expressions `A` and `I` respectively. But placing a list of values in the code, or a list of expressions to compute certain values, is already done for initializers. Since, from the user viewpoint, indexed and array selections are very similar, we prefer to keep them together in the feature test macros. Therefore, support for direct selections is included together with support for indexed selections when `__STDC_ARRSEL_STEPPED__` is `2`.

`__STDC_ARRSEL_NESTED__`: A value of `0` here does *not* mean that array or direct selections can only be one-dimensional. The added complexity of two- or higher- dimensional range selections is that a broken second (or inner) dimension can arise. Hence, the address of `A[0][0]` might not be the same as that of `A[0]`; also, `A[0]` will not be a "plain" array.

By contrast, a selection {*e*,*f*} collapses the first two dimensions of `A` into one and selects there the element *e*·*l*+*f*, where *l* is the length of `A`; this is the element at offset *e***s*+*f***s*' from the base of `A`, where *s* is the size of each (array) element of `A` and *s*' that of `A[0]`'s elements. These computations of offsets are right provided `A` is not carrying a broken selection, and in particular if it is not carrying a selection, which is precisely what **__STDC_ARRSEL_NESTED__** guarantees.

Nor is there a need for two indices when translating range operations:

```
T A[n]; int I[n];
A[I[:]] = B[:]+C[:];
for i=1..n  A[e(i)]=B[i]+C[i]
```

```
T A[n][l]; int I[n][2];
A[I[:]] = B[:]+C[:];
for i=1..n  A[e(i)*l+f(i)]=B[i]+C[i]
```

where *e*(*i*) at the left stands for the *i*-th element of `I` and *e*, *f*, *l* at the right have the same meanings as above, and `A` there stands for the matrix at the address of `A` interpreted with type *T* `A[`*n***l*`]`.

What a value of `0` in this macro would mean is that indexed and direct selections can only be applied to the same expressions as a `[B:L]` selection; i.e., expressions of pointer type or arrays not carrying a selection or carrying an empty selection.

__STDC_ARRSEL_CONSTANT__: If it is `1`, only **`constexpr`** arrays are allowed as the indices, or compound literals where all the elements are integer constant expressions, or strings. For direct selections, it restricts the integers therein to ICE.

Some intermediate combinations could be possible, such as requiring a fixed length array as the index. But as we argued, it is better to restrict the possible combinations of partial implementations. Note that a value of `1` in this macro also implies that the length of the selection carried by the index matrix, if it carries one, is given by an integer constant expression.

Comma-separated list

Mixing with CRE

It may be preferred to keep a comma-separated lists of integers and constant range expressions separated, so that

```
A[3,6,5,2,1], A[3...10]
```

would be allowed but `A[0, 3...10, 13]` not. Since a CRE could always be expressed by a (possibly long) comma-separated list of integers we prefer to allow the mixing; that is, a CRE in the middle of a list is not different than many single values in its place.

However, a translator can treat a lone CRE as the equivalent `[B:L]` selection, and may place the values of a comma-separated list with no CREs in the generated code (or just reserve the space for those that need to be computed at runtime) and use that in-memory list to take the indices from there at runtime. This second strategy will fail or become inadequate if a range expression specifying a long range is inserted in the list:

```
A[0,1, 0...99'999, 99'998,99'999]
```

A long list may also arise as the result of a use of **`#embed`**. In order that the implementation can store all the integers in memory, we allow it to place a limit on their number:

— 32767 for the total number of bytes required to store the subscripts specified in a direct array selector list (in a hosted environment only), counting each number in a

constant range expression individually.

which copies the limit for the total number of bytes in an object. We'd like to see a similar but substantially smaller limit for freestanding environments, but since there is none currently expressed for the number of bytes in an object, we don't express one here either.

Whether the numbers within each `{}` in case of bidimensional lists, equal in number to the number of dimensions in which the selection acts, have to be stored individually or it is possible to just store the resulting index in the collapsed matrix, depends on whether all quantities involved are integer constant expressions or not: those numbers and the sizes of the arrays by which they have to be multiplied. These bidimensional lists cannot contain CREs, so we don't think any program will be restricted on that side.

An implementation may be able to handle lists mixing constant range expressions and individual values without having to store all the values in the range expression, but is not required to.

Conflict with array subscripting

We have designed 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 be include the comma operator. That is, currently `A[3, 0, 1, 5]` is interpreted as `A[5]`. There are several solution for this.

1. To add an extra pair of `{}` enclosing the whole list. This is the most obvious and straightforward solution.

   We applied this while preparing this document. This proved cumbersome, specially for multidimensional selections, as in `A[{{1,2},{2,3},{3,3}}]`. Mandating them just for one-dimension selections is not possible for then a selection like `A[{1,2}]` becomes ambiguous. It 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.

   We think the definition of such a directive would be very problematic. What if an identifier is declared in "new" mode using some new feature and then used in the "old" mode, for example? However, a directive that changes the way the program is translated *only for selected features* is easy. For example

   ```
   #STDC_features 29
   ```

   If the value is < 29 the comma is interpreted as the comma operator and direct selections other than a CRE are not allowed; otherwise, the expression k in `[`k`]` for the subscripting operator is a conditional expression. this way old files can be compiled with `#STDC_features` set to some value < 29 and new code with it set to 29.

   In addition it can change the way `I` is treated, or if `0[`p`]` is allowed, for example.

   This directive would have an implementation predefined value ≤ 29 (that compilers would control via an invocation flag)

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

   ```
   #STDC_features ARRAY_COMMA 0
   ```

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

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

The fourth and fifth solutions have the drawback compared with the two previous ones that the state cannot be changed within the translation unit. But this may be felt like an advantage rather than a drawback. Likewise, the directive from solutions 2 and 3 might be limited to apply to whole translation units. The introduction of the directive from solution 2 is being discussed presently.

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 cannot place a comma operator inside an array subscript seems a mild requisite and even desirable.

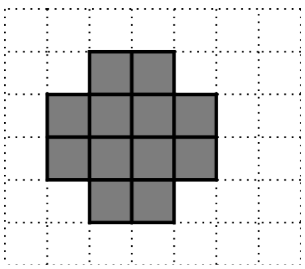
A comma operator only makes sense if the first operand has side effects, which are not allowed in direct array selectors. Hence, in solution 4, compilation with `__STDC_ARRAY_SELECTIONS__` defined to `1` will make those expressions trigger an error (and similarly for solutions 2, 3 and 5) The safest path towards the elimination of those expressions is as follows: first, the code is compiled without array selections (using any version of C older than C2y or using C2y with a switch to turn array selections off) and asking the compiler to issue a warning for discarded expressions with no effect. This will detect dummy comma operators. These are likely to be bugs in the code. These are fixed. Then the code is compiled with array selections. Comma expressions within `[]` with side effects will cause a compilation error. For these, the subscript is enclosed in `()`. E.g., `[(p++,n)]`.

In the event that direct selections get added by some implementation, that implementation will likely provide a command line switch to interpret the comma as list separator. This is the fifth solution. Likewise, when that switch is off, it should issue warnings for any comma operator inside `[]` which is not hidden by braces (i.e., when the expression inside `[]` is proper comma expression). This may be the practical way towards commas inside `[]`. It is unlikely that users will not want to get rid of the extra `{}`, and that command-line switch will thence get standardised as an implementation defined behaviour (for which a testing macro will be provided).

Furthermore, array subscripting including a comma operator is extremely unlikely to appear in header files; there may be none at all, and there will be no problem in compiling in the new mode including files from older libraries (plus, do not forget, that a compiler will very likely provide a warning for commas within `[]` when compiling in the old mode).

Example of use of indexed selection

Let us suppose we have a large matrix of size $n \times n$. Within it, a small shape moves:



The shape operates on the elements of the matrix it lies onto.

We begin by defining the pixels that form the shape with respect to its upper left corner:

```
const int Sh[][2]={      {0,1},{0,2},
                   {1,0},{1,1},{1,2},{1,3},
                   {2,0},{2,1},{2,2},{2,3},
                         {0,1},{0,2}
                  };
```

Next, suppose we have some value `n` giving the size of the matrix and an initial position, at `r0`, `c0` within that matrix:

```
unsigned char A[n][n];
int pos[2]={r0,c0};   //Position: row and column
```

Finally, we move the shape along a diagonal of slope -1/2 two hundred times, performing a certain operation at each position:

```
for(int k=200; k!=0; k--){
    A[pos[]+Sh[:]] ^= 0x3F;
    pos[0]++, pos[1]+=2
    A[pos[]+Sh[:]] ^= 0x3F;
}
```

Note that here `pos[]+Sh[:]` is defined by the general rule for the 3rd case in “Arrays with selections of different depth”.

11 Our final choice for the type

As can be seen in the forgoing pages, the short interpretation always gives simpler semantics, but has the problem of the very different memory layout of an array with selection as compared to the corresponding array without selection. In the long interpretation, on the contrary, the memory layout is the same, which hints at that being the right type; the difference being only that some elements are selected, other not. But that is not true, as shown by the next section. For this reason; the situation of selection from an array of pointers; the value which we like `_Lengthof` to yield for these arrays, and that it presents itself to simpler solutions, we finally chose 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;
{𝑠, 𝑖, 𝑖, 𝑠, 𝑖, 𝑖, 𝑠, 𝑖, 𝑖, 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 consideration of the ignored elements as *ignored;* i.e., belonging to the array but omitted from the operation about to take place, can still be kept notwithstanding the previous analysis, but we prefer to consider those bytes as being outside the object.

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.

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 term is introduced to keep the type system consistent.

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? In the long interpretation it was clear that the term could only apply to those actually broken, since that is the meaning it conveys in that interpretation (for there it is not a qualifier); e.g., only unbroken arrays are allowed in sizeof, in that interpretation.

In the short interpretation the situation is different. 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 would mean 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, which in the short interpretation means that theirm memory layout is not the same as that of a plain array, are deemed broken. If the qualifier were introduced in the 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.

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.

12 Further extensions

The extensions treated in this section arise naturally. There inclusion here does not mean that the author is in favour of their eventual adoption.

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]`. If so, this could be incorporated to the standard in the future.

Range selection constraint related recommended practice

Constraints

Recommended practice

For the form `[B:L:s]` in case the postfix expression is a complete array which is not a variable length array, `L` is an integer constant expression and `s` is not, implementations are encouraged to produce a diagnostic message in case `L` is greater than the length of the array or, if also `B` is an integer constant expression, both `B+L` and `B-L` fall outside the range of valid indices for the array. These expressions can only have defined behavior if `s` evaluates to zero.

The rationale for this is that a step which is not an integer constant expression likely takes different values on different evaluations. Also, that a range selection which can only be valid if *s* is zero should be written with a literal 0 in place of `s`, or at most an i.c.e with value 0.

Relaxing the UB of overlapping in assignments

4 If the left operand of an assignment expression is an array, for each singleton *i* of it in which a value is stored let C(*i*) be the set of singletons that need to be read, in the expression at the left 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. (Option 1) the object representation of *i* becomes unspecified.

Or even more relaxed:

(Option 2) ... another singleton *j* ... the object representation read for *j* for the computation of *i* is unspecified.

The latter is more relaxed as shown by the following examples:

EXAMPLE

```
unsigned char A[10];
A[0:9] = 3 + 0*A[1:9]; //A[0:9] becomes unspecified in option 1. But in
                       //option 2 the unspecified values read from A[1:8]
                       //are irrelevant for result of the operation.
```

After the following expression

```
A[0:9] += A[1:9];
```

all values in the range `A[0:8]` become unspecified while `A[8]` is well defined, in both options. Also, after

```
A[2] = A[1] = A[0] = 1;
A[0:2] = A[1:2];
```

the value of `A[0]` becomes unspecified, for while the expression does not modify `A[1]` a value is stored in it. `A[0]` equals 1 in both options.

EXAMPLE The unspecified value read for *j* may lead to undefined behavior:

```
int A[3]:
A[0]=0; A[1]=1; A[2]=2;
A[0:2] /= A[1:2];
```

Here, in option 2, the value read for `A[1]` for the computation `A[0]/A[1]` is unspecified, and if it is zero the behavior is undefined. Option 1 here is more restricted towards the possible outcome: `A[0]` becomes unspecified but the program does not have undefined behavior.

We prefer option 1, which does not force the translator to compute a correct value when overlapping occurs in cases like `A[0:9] = 3 + 0*A[1:9]` above. But the last example shows that, if option 1 is chosen, lee in the reading of *j* is also needed, so as not to force a defined behavior (though resulting in an unspecified value) when, e. g., division by zero may occur. Considering this, the text we propose combines the freedom given to the compiler of both options (i.e., is the strictest for the programmer):

the object representation read for *j* for the computation of *i* is unspecified and the object representation of *i* becomes unspecified.

We like this constraint in the possible behaviour of the program, with respect to the current undefined behavior of this proposal, for it removes an unnecessary u. b., limiting it to an unspecified value in most cases, which is what the emitted instructions could possibly do. It may be that an allowance for unspecified representation of all the computed elements, not just those involved in the overlapping, is necessary. Implementation experience will tell.

Relaxing the restriction for overlapping

The restriction for overlapping can be partly lifted, if it be so desired, to allow either or both of

```
A[0:9] = 2*A[3];
```

```
A[0:9] = f(A);
```

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

In a selection of the form `A[B:L]` where `A` is not an array with selection the resulting object is the same whether `A` has array or pointer type, both in its memory layout and in its type:

```
int A[10], *B;
A[2:5]; B[2:5]; //Both are an array of 5 int.
```

But if `A` is itself an array with selection the situation is different:

```
int A[10][6], *B[10];
A[2:4][0:3];   //[a,b,c,i,i,i,  a',b',c',i,i,i,  a'',b'',c'',i,i,i,  a''',b''',c''',i,i,i]
B[2:4];        //Array of four pointers
B[2:4][0:3];   //[a,b,c]  [a',b',c']  [a'',b'',c'']  [a''',b''',c''']
```

Here *i* represents an ignored (not selected) element. In the last line we've got four arrays, at the addresses pointed to by `B[2]`, `B[3]`, `B[4]` and `B[5]`. The memory layout is very different form `A[2:4][0:3]` and the four arrays may not even lie in the same storage instance.

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.

Long interpretation

The definition of a type for this kind of selection before lvalue conversion is difficult. We may say it is an "array with selection of sparse type". Then make the type of any instance of this incompatible with any other type, including other sparse selections, and insert appropriate wording for the assignment and `++`, `--` operators. Obviously, these cannot be operands to **`sizeof`** and should not be to **`typeof`**.

Short interpretation

The type and value returned by **`sizeof`** are the same as for any other broken array.

Selections [], [:] and [::]

The selection `[:]` has no meaning when the element it applies to is a pointer. This is forbidden in the wording and needs no further mention for this case.

`[]` has no effect, as when applied to any array already carrying a selection.

`[::]` seems even more meaningless than `[:]`, but it does have meaning. That selection means to select all dimensions up to the singletons. Pointers are singletons, so it cannot select any further; its meaning is the same when applied to an array with selection where the selection has already reached the singletons, whether the singletons are of pointer type or not. We allow it when there are no more dimensions from which to select, an pointers make no difference in this respect:

```
float A[9][9], *B[9][9];
A[::], A[:][::], A[:][:][::];
B[::], B[:][::], B[:][:][::];
```

The first selection from each line selects two dimensions; the second one, one; the third one, none (`[::]` has no effect).

typeof, sizeof, _Unselect() and _Value()

Long interpretation

The restrictions which apply to arrays with selection in **typeof** and **sizeof** have been justified. We'd rather add **`_Unselect()`** and **`_Value()`** operators than lift those restrictions. It must be clear whether the user wants the type / size of the whole matrix or just that of the selection:

```
int A[10][10], B[20], n;
typeof(_Unselect(A[2:3][0:5]))   // Clear. typeof(A[2:3]).  int[3][10]
typeof(_Value(A[2:3][0:5]))      // Clear. typeof(int[3][5])
typeof(A[2:3][0:5])              // ?
typeof(_Unselect(B[2:n:4]))      // typeof(B[2:1+4*(n-1)]).  int[4n-3]
typeof(_Value(B[2:n:4]))         // typeof(int[4])
typeof(B[2:n:4])                 // ?
```

`_Unselect()` makes visible whether margins are retained or not:

```
typeof(_Unselect(A[2:3][0:5][0])) // int[10], if margins are kept.
typeof(_Unselect(A[2:3][0:5][0])) // int[5], if margins are not kept.
```

When used for reselecting, recovering all the elements of the full array is probably not a good choice:

```
#define invert3(x) _Unselect(x)[0:3] = 1/_Unselect(x)[0:3]
```

The macro probably does not work as intended for broken arrays. It does not select the first three elements from `x`, but the first three of the corresponding full array

Short interpretation

sizeof is unambiguous, as is **`typeof_unqual`**. **`typeof`** should return the qualified type in the event that the **`selected`** qualifier is added to the language, and in the meantime (forever?) arrays with selections are not allowed there. Therefore, **`_Value()`** is of no use for these operands.

`_Unselect()` cannot be applied to a broken lvalue in the short interpretation, because the memory layout of the array is different than what the result of **`_Unselect()`** should be:

```
#define invert3(x) _Unselect(x)[0:3] = 1/_Unselect(x)[0:3]
```

The second **`_Unselect()`** could be fixed by preceding it with **`_Value()`**: **`_Unselect(_Value(x))`**, but that is not possible for the first one.

The solution is to split the property of carrying a selection from the brokenness qualifier. Thus, **`_Unselect()`** applied to an lvalue would remove the selection but the resulting arrays would still be broken-qualified if it was so.

`_Unselect()` makes translation more difficult. In the absence of it, the translator knows it can translate selections into for loops. An un-selection complicates it by making a subsequent selection apply to the already broken array:

```
_Unselect(A[0:8:3])[2:4]+B[0:4]   //Operates A[6], [9], [12] [15].
```

Functions taking and returning arrays

Now that there is a way of preventing a matrix to decay to a pointer, functions may be declared to take arrays as argument, as in the following example:

```
double determinant3(double A[:3][3]){
    double d;
    /* ... */
    return d;
}

double M[3][3], N[6][6];
determinant3(M[]);
determinant3(N[0:3:2][0:3:2]);
```

The obvious choice for the parameter and, we would say, the right choice, is `double A[3][3]`, which we all know is not possible.
As for the returned value, now arrays make sense because they can be assigned to:

```
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[]);
```

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

```
double invert3(double A[:3][3]) [3][3];
```

We would like the parameter as well to be declared by means of the defined type:

```
dbl33 invert3(dbl33 A);
```

but as we noted this is not possible.

Functions acting as range operators

Suppose we want to compute the square root of all the elements on the main diagonal of a matrix. Writing

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

obviously doesn't work. What is needed is a way of saying that the function has to act on

each of the selected values. We thought of the following syntax for it:

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

The semantics is that `[]` after the function name means that there are as many invocations of the function as there are selected elements in its arguments which are arrays with selection (the selections in the different arguments must match). All other arguments get evaluated once. Examples with more than one argument:

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

The result is an array with selection; that is, the expression has array type and carries a selection. The selection is of the form of that of the arguments (in the long interpretation, after lvalue conversion), which must be identical for all of them, and the type of the singletons 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])
A[::] = exp[](Ω[::])
```

There is no concurrence with the use of the `[]` operator as empty selector, since the identifier used to call a function may be a function pointer but not an array of function pointers. For such an identifier as the latter `[]` retains it normal meaning, and a list of arguments cannot follow.

Address to broken arrays. The broken qualifier

This is a substantial extension.

Suppose we take the address of a broken array:

```
int A[15], B[6][8];
int (*p)[broken 5]= &A[0:5:3];
int (*q)[broken 6][4]= &B[:][0:4];
int (*r)[broken 6][broken 4]= &B[:][0:4:2];
```

We need the qualifiers, in some form, because `p`, `q` and `r` do not point to “normal” arrays of integers. For example, `A[0:5:3][1]` is not at the address following `A[0:5:3][0]`.

If the address of the broken arrays cannot be taken, that broken object can get nowhere, and the translator only needs to know its layout for the translation of the expression where it appears. By taking its address and assigning it to an identifier, now that address may be passed around. The following should not be allowed:

```
int (*pp)[5] = p;
```

since `(*pp)[1]` will not access the right address. If we allow the pointer to be copied and passed to functions, the full introduction of the qualifier in the type system is unavoidable.

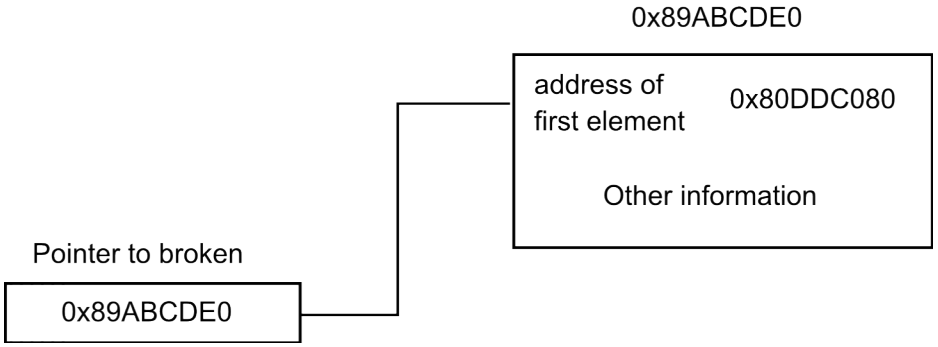
If the **broken** qualifier is introduced it will apply to arrays, not to pointers, but is most meaningful when using pointers to broken arrays:

```
int (*p)[broken 6]= &A[0:6:3];
```

```
f(p);
```

Different **broken**-qualified arrays will in general have different layouts. Therefore, the function `f`, when receiving the argument `p`, needs to get with it the information of how the broken array is actually broken. Of course, the six elements could, in particular, be stored consecutively, so **broken** should mean potentially broken.

The amount of information pointers to broken arrays need to carry means that they cannot be represented as the corresponding unqualified pointer. A possibility is that the pointer points to a block of memory holding the information of how the array is stored in memory and, in particular, the address of it:



This way the size of the pointer is still that of a normal pointer.

Broken-qualified arrays should not decay to pointers. And since broken-qualified means potentially broken, **broken** could be called **selected** instead. That would make potentially-broken and carrying-a-selection synonyms. But we may keep the two concepts split, so that a broken-qualified array need not necessarily carry a selection:

```
int B[6][8];
int (*q)[broken 6][4]= &B[:][0:4:2];
(*q)[0:2];  //B[0:2][0:4]
q[0][1:2];  //B[0][2,4]
```

In this design, taking the address forgets the selection. This is the most versatile behaviour.

Upward propagation of broken

As we already observed, in the short interpretation, if an array's elements are broken so has to be considered the array itself. The following type names therefore denote the same type:

```
int (*)[6][broken 4]
int (*)[broken 6][broken 4]
```

If these types were to be different, the first one would imply that the six elements are stored in order. But to know the position of the next element knowing the position in memory of the four pieces that compose the first element requires remembering the original full array, and this is the long interpretation. E.g.,

```
int A[6][12], B[6][7];
&A[:][0:4:2];
&B[:][0:4:2];
```

Since `A[:][0:4:2]` and `B[:][0:4:2]` have different layouts even if `A[0]` and `B[0]` are equal, only one of them, and of the infinte many possibilitys having the same layout of its `[0]`-th element, could be the one with type `int [6][broken 4]`. We could say that it is the one where the last element in memory from the `[0]`-th element immediately precedes the first element in memory of the `[1]`-th element, but it seems there is no reason for signalling out that one in particular.

Collapsed arrays

Broken-qualified means potentially broken. Thus, `A[1]` may not be at the address just following `A[0]`. But in many cases it will be true that `A[1]` is not at the same address as `A[0]`. Since that is not guaranteed in general in a broken array, it might be good to split that property to a separate qualifier: **`shared`**.

```
double func(int (*A)[broken 6]){
    /* ... */
    (*A)[k]= a;
    (*A)[k-1]= b;
}
```

In this function body, the compiler can be sure that reordering of the two assignments is safe, since the array cannot be collapsed. On the other side,

```
double func(int (*A)[shared 6]){
    /* ... */
    A[k]= a;
    A[k-1]= b;
}
```

Now reordering is not possible.

`shared` means, in particular that the layout of the array's elements is not like that of a plain array. Therefore, **`shared`** implies **`broken`**.

The first situation that can give rise to a shared array is a selection with step zero, where all elements share the same space in memory. For this reason we first called the attribute "collapsed". But irregular selection (indexed and direct ones) can create arrays where some but not all of the elements share the space in memory, so "shared" seemed better.

Pointer hierarchy

A broken-qualified array may or may not be broken; a regular array can never be broken. Thus, a pointer-to-broken is a supertype of a pointer-to-plain, in the language of the proposal "Enhanced type variance". Likewise, a pointer-to-collapsed is a supertype of a pointer-to-plain.

13 Further editorial fixes

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, as it often happens in the standard. 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.

6.5.17 Assignment operators (again)

The paragraph restricting the overlapping of the assignee with the read object 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.

14 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. Dark blue text is new text for the option a zero step is allowed, dark yellow is new text for the option a zero step is not allowed. We noted that we do not consider this latter option.

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; B for an array with no selection or empty selection, and s for a singleton.

6.2.5 Types

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.

6.3.3.1 Lvalues, arrays, and function designators

1 An *lvalue* is an expression with an array type or a complete object type 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

— has array type and its singletons are modifiable lvalues

3 Except when it is the operand of the **sizeof** operator, or the typeof operators, the unary **&** operator, the **++** operator, the **--** 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 selected elements" because the selected elements need not be singletons and "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 Except when it is the operand of the **`sizeof`** operator, or the **`_Lengthof`** operator, or the typeof operators, or the unary **`&`** operator or one of the two expressions of an array subscripting operator, or the left operand of a range selection operator, or the left operand of an assignment, or is a string literal used to initialize an array, an expression that has type "array of *type*" and the array does not carry a selection is converted to an expression with type "pointer to *type*" that 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 (6.5.3.2), range selection (6.5.3.3), postfix increment and decrement operators (6.5.4.6), prefix increment and decrement operators (6.5.5.2), the **`sizeof`** 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 operators

Constraint

1 For the multiplicative, additive, bitwise shift, inclusive or, exclusive or and 'and', equality, relational and assignment operators, if the two operands are arrays with selection then: If the selections they carry are nonempty and their lengths are given by integer constant expressions, these lengths shall be the same. If the innermost selected elements of both operands are arrays, the number of dimensions of the latter, considered as multidimensional arrays, shall be the same; for these arrays, if for any dimension the length in one and the other operand is given by an integer constant expression, they shall be the same. Excepting relational and equality operators, if one of the operand carries a selection of singletons (i.e., with selection in as many dimensions as the array has), so shall the other.

Semantics

2 If the two operands of the operators listed in the constraint are arrays carrying a non-empty selection their lengths 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 the carried an empty selection (that is, they are not converted to pointers). A[:] *op* A[:]

3 For the same operands as in the previous paragraph, if one operands is an array carrying a selection of singletons , and the other operand is a singleton or an array which is converted A[:]s *op* s

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.

4 For the same operators as above with the exception of relational operators, if one operand is an array without selection which is not converted to a pointer (e.g., as a result of the application of the rule in p. 2) or an array with selection where the innermost selected elements are of array type, so shall be the other operand, and the innermost selected elements from one and the other operand (or the operand itself if it does not carry a selection) shall be arrays with the same dimensions (considered as multidimensional array if their element type is an array type). In this case, if one of the operands carries no selection or an empty selection, let `E` be the other operand. For the operators considered with the further exception of equality operators, 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 each innermost selected element from `E` with the corresponding singleton of the other operand; each singleton of this latter operand is evaluated only once.

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

5 Whenever an array has each of its elements operated with the other operand, or with a corresponding element of the other operand, the order in which the operations are performed is unspecified.

6 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.

7 **EXAMPLE** 1

```
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]`, as a result of the lvalue conversion of `B[0:6]`. 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`]`.

8 **EXAMPLE** 2

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

The rule in p. 2 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. 3 applies to it; the result of each operation is the array `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).

9 **EXAMPLE** 3

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

```
B[:] *= C[];
A[:] *= D[];
A[:][:] *= B[:][];
```

The rule in p. 4 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. 2 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`).

10 **EXAMPLE** 4 The code

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

has undefined behavior. For 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.

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:
[**:** **:**$_{\text{opt}}$ **]**
[*conditional-expression* **:** *conditional-expression* **]**
[*conditional-expression* **:** *conditional-expression* **:** *conditional-expression* **]**

6.5.4.2 Array subscripting

The text of this section is to be that of the (hopefully) eventually approved proposal "Array subscripting without decay".

6 **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 `[::]` 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, that of `[B:L:s]` to select the elements at positions `B`, `B+s` ... `B+(L−1)*s` and that of `[::]` to select all elements from all dimensions. The order in which the expressions `B`, L and `s` are evaluated is unspecified.

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 array selector is of the form `[::]` and the postfix expression is an array with selection, the array selector of the latter, and that of its postfix expression if this is an array selection, and so recursively, cannot be `[::]`.

6 In any form of the array selector except `[::]` and `[]`, if the postfix expression 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.

7 If `L` is an integer constant expression it shall be greater than zero. If `s` is an integer constant expression it shall be nonzero.

8 If 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 and is not a variable length array, 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, where l is 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.

Semantics

9 In the following paragraphs, let b, l and s be the result of evaluating `B`, `L` and `s` respectively. b shall be nonnegative and l shall be greater than zero. No restriction applies in general to s; in particular, it may be zero. , and s shall be nonzero.

10 An array with a memory layout different from that of an array without selection will be called a *broken array*. A broken array behaves as if it had a *brokenness* qualifier. This property is lost upon lvalue conversion. Different broken arrays that have the same type may have different memory layouts. Conditions under which an array with selection is broken are stated in the following paragraphs.

11 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 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) el-*

ements selected. Let *type* be the type of `A[0]`. The expression has type "array of *type*". It has length *l*. If `L` is an integer constant expression it is an array of known 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 `A[`*b*`]` is the first, `A[`*b*`+1]` or `A[`*b*`+`*s*`]` the second and `A[`*b*`+`*l*`−1]` or `A[`*b*`+`*s*`*(`*l*`−1)]` the last. A selection of the form `[B:L:s]` where neither *l* nor *s* are 1 is a *stepped selection*. An array carrying a steped selection is broken.

12 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, and the type of the innermost selected elements is *T*, then: If *T* is not an array the behavior is undefined. Otherwise, let `A` have an *m*-dimensional range of elements selected, of type *T*, and *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 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*≠0 or *l*≠l_v the arrays *u* in `A[B:L]` or `A[B:L:s]` are broken.

13 If the elements of an array are broken arrays, the array itself is broken.

14 In an expression of the form `A[:]`, let *x* represent `A` if this latter array does not carry a selection, or its innermost selected elements if it does carry a selection. The expression is equivalent to `A[0:`*l*`]`, where *l* is the length of the array(s) *x*, except that if (each) *x* is a top-level variable length array so is (each of them) in `A[:]`. (That is, the expression is equivalent to `A[0:_Lenghtof(`*x*`)]`).

15 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 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.

16 If the expression is of the form `A[::]` and `A` is an *n*-dimensional array with no selected elements or, if it has selected elements, the innermost ones of these are *n*-dimensional arrays, then: If no array selectors follow the `[::]`, `A[::]` is equivalent to `A[:][:]` ... `[:]`, where there are *n* `[:]` selectors; if `A` is an array with selection *n* may be zero. If array selectors follow (none can be of the form `[::]`) and among these the range selectors are *m* in number, `A[::]` is equivalent to `[:][:]` ... `[:]`, where there are *n*-*m* `[:]` selectors; *n*-*m* may be zero.

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.

17 **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 singletons of type `int`. `x[0:1][:]` is the same array and has selections from both its dimensions, resulting in a two dimensional range of elements selected, each of which is a singleton of type `int`.

18 **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 variable length array while the latter is not. `z[0:2]` has type array of two `int`. `z[n-3:1]` has type `int[1]` and is not a broken array.

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

19 **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]}`

20 **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.

21 **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]`.

22 **EXAMPLE** 6

```
int A[6][6][6];
A[::];                  //Equivalent to A[:][:][:]
A[::][0:3];             //Equivalent to A[:][:][0:3]
A[::][0:3][:];          //Equivalent to A[:][0:3][:]
A[::][];                //Equivalent to A[:][:][:]
```

6.5.4.6 Postfix increment and decrement operators

Constraints

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.

Semantics

2 If the left operand is an array with selection it shall not carry a stepped selection where the step is zero, nor shall its selected elements and so recursively.

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

4 The *adjustment value* is the value used to increment or decrement the value of the operated elements. If the operated elements have pointer type, the adjustment value has type `int` and the value `1`; if the operated elements have complex type, the adjustment value has the

corresponding real type of the operated elements and the value `1`; if the operated elements have decimal floating type, the adjustment value has the same type as the operated elements, `1` as the numerical value, and `0` as the quantum exponent; otherwise, the adjustment value has the same type as the operated elements and the value `1`.

5 The result of the postfix `++` operator is the value of the operand. As a side effect the value of the operated elements is incremented by the adjustment value. 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 indeterminately sequenced 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.

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

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.

Semantics

2 If the left operand is an array with selection it shall not carry a stepped selection where the step is zero, nor shall its selected elements and so recursively.

3 The value of the operated elements (6.5.4.5) of the prefix `++` operator is incremented. The result is the new value of the operand after incrementation. The expression `++E` is equivalent to `(E+=1)`, where the value `1` is the adjustment value (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 an array subscripting operator or a unary `*` operator, or an lvalue that designates an object that is not a bit-field. This lvalue or, in case the operand is the result of an array subscripting operator or a unary `*` applied to an lvalue, this latter lvalue, shall not designate an object 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 `_Lengthof` 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).[101] 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 following equality always holds for arrays:

```
sizeof(array) = _Lengthof(array)*sizeof(array[0])
```

6.5.6 Cast operators

Constraints

2 One of the following shall hold:

— The type name specifies a void type; or

— the type name specifies an atomic, qualified, or unqualified scalar type and the operand has scalar type or is an array with no selection or carrying a selection of singletons.

— the operand is an array with an empty selection and the type name specifies an array type with the type of the singletons as an atomic, qualified or unqualified version of a type compatible to that of the operand's singletons.

3 If its operand is an array of fixed constant total length carrying an empty selection and the type name specifies an array of fixed constant total length, the total length of the latter shall be less than or equal to the total length of the operand.

Length and not size, though since the singletons' types are compatible, there is no difference.

[...]

Semantics

[...]

6 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*.[102] 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 conversion is applied to each of its singletons. A cast that specifies no conversion has no effect on the type or value of an expression.

Each element from the result comes from one element from the operand.

7 If the operand is an array with selection then: if the type name specifies an array type, the total length of this type shall be less than or equal to the total length of the operand and

the value of the expression is an array with an empty selection; otherwise the array carries a selection of singletons and 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.

[...]

6.5.7 Multiplicative operators

[...]

Constraints

2 Either operand may be an array with selection. Its singletons must 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. If so, the other operand cannot be an array without 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.

Semantics

6 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.

[...]

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

[...]

14 **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.

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. If so, the other operand cannot be an array without 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 one operand is an array without selection the other operand cannot be an array carrying a selection. If either or both operands are arrays not carrying a selection they are converted to pointers to their first elements and it is to these pointers to which the condition 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. If the other operand has pointer type or type **`nullptr_t`** so is *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.

Semantics

6 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.

7 [...]

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

9 [...]

10 [...]

11 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 larger 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. sel-A *eq* s

12 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 larger array), the dimensions of the selected arrays *a* and the array `B` shall be the same. 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. sel-A *eq* B[]

13 **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.

16 **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.

17 **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.

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. If for any dimension the length in one an the other operand is given by an integer constant expression, 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 operand 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 the second and third 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 EXAMPLE [...]

12 If both operands are arrays with selection their dimensions (considered as multidimensional arrays if their element type is an array type) shall be the same. The common type is determined by applying the above rules to the types of their singletons.

6.5.18 Assignment operators

6.5.18.1 General

[...]

Constraints

2 An assignment operator shall have a modifiable lvalue as its left operand. If the right operand is an array with selection the left operand shall be an array. If the left operand is an array with an innermost selection of singletons, then the right operand shall be a singleton, or an array without selection or an array carrying an innermost selection of singletons.

3 If the left operand is an array with selection, the latter shall not be given by a selector of the form `[B:L:s]` where `s` is an integer constant expression of value zero, nor shall any of its selected elements have such a selection and so recursively.

`s` an ICE =o could be allowed with a non-ICE `L`, but it seems better like this.

[...]

Semantics

5 If the left operand is an array with selection it shall not carry a stepped selection where the step is zero, nor shall its selected elements and so recursively.

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 with an innermost selection of singletons, these together with the right operand if not an array with selection or the selected singletons of the right operand if it is an array with selection must 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 carrying a selection where the innermost selected elements are of array type, then so is the left operand or it is an array with no selection. 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). In this case, the singletons from which the left and right arrays are composed shall satisfy the constraint above for operands which are not arrays and, in addition, if these singletons from the left operand have pointer type or type **`nullptr_t`**, so shall have the singletons from the right operand.

Semantics

4 If the left operand is an array without selection the expression is equivalent to one in which the left operand has had the empty range selection `[]` applied to it.

[...]

6 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 as-

signment 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.

8 EXAMPLE 1 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]`.

9 EXAMPLE 2 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.

[...]

13 EXAMPLE 5 Assignments where the left operand is an array.

```
int A[5][4], B[5][4], C[4];
A[:] = C[];           // Equivalent to A[0][:]=C[:], A[1][:]=C[:], etc.
A[:] = B[0][];        // A[0][:]=B[0][:], A[1][:]=B[0][:], etc.
A[:] = B[:];          // Constraint violation
A[:][:] = B[:][:];    // Write this instead
A = B[];              // Equivalent to A[:][:] = B[:][:]
A[0][:] = C;          // Implicit conversions from size_t to int
```

Writing `C[]` and `B[0][]` above, instead of `C` or `B[0]`, is necessary to prevent the array decaying to a pointer.

6.5.18.3 Compound assignment

Constraints

1 For the operators **+=** and **−=** only, if the left operand is not an array with selection then: either the left operand [...]

2 For the other operators, if the left operand is not an array with selection then: the left operand [...]

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

4 If either operand is an array with selection, the innermost selected elements shall be singletons and shall satisfy the constraints set forth in the previous paragraphs.

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

4 [...] (The declaration of variable length arrays with automatic storage duration are a conditional feature that implementations may support; see 6.10.10.4.)

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_ARRAY_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.