# Restricted Pointers
# WG14/N448
# X3J11/95-049

Bill Homer
Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121

August 23, 1995

## 1 Introduction

This chapter describes a proposal for an additional type qualifier, spelled `restrict`. Like the `volatile` type qualifier, `restrict` is intended to affect compiler optimizations (though in the opposite direction, by enabling rather inhibiting them). Like the `const` type qualifier, `restrict` has semantic rules to which a program must conform to avoid undefined behavior. The proposed changes to the text of the C Standard, which are relatively small and confined to two sections, are presented after the rationale.

## 2 Rationale

### 2.1 Aliasing

For many compiler optimizations, ranging from simply holding a value in a register to the parallel execution of a loop, it is necessary to determine whether two distinct lvalues designate distinct objects. If the objects are not distinct, the lvalues are said to be *aliases*. It is aliasing through pointers that presents the greatest difficulty, because there is often not enough information available within a single function, or even within a single compilation unit, to determine whether two pointers can point to the same object. Even when enough information is available, this analysis can require substantial time and space. For example, it could require an analysis of a whole program to determine the possible values of a pointer that is a function parameter.

### 2.2 Library examples

Consider how potential aliasing enters into implementations in C of two Standard C library functions. There are no restrictions on the use of `memmove`, and the implementation shown in Figure 1 follows the model described in the Standard by copying through a temporary array. (Other approaches are possible, but this one is straightforward and strictly conforming.) Since `memcpy` cannot be used for copying between overlapping arrays, its implementation can be a direct copy, as in Figure 2. (See the appendix on "Overlapping objects.")

Note that the restriction on `memcpy` is expressed only in its *Description* in the Standard, and cannot be expressed directly in its implementation in C. While this does allow the source-level optimization of eliminating the temporary used in `memmove`, it does not provide for compiler optimization of the resulting single loop. In many architectures, it is faster to copy bytes in blocks, rather than one at a time. The implementation of `memmove` uses `malloc` to obtain the temporary array, and this guarantees that the temporary is disjoint from the source and target arrays. From this a compiler can deduce that block

Figure 1: Sample implementation of memmove.

```
void *memmove(void *s1, const void *s2, size_t n) {
    char *t1 = s1;
    const char *t2 = s2;
    char *t3 = malloc(n);
    size_t i;
    for(i = 0; i < n; i++) t3[i] = t2[i];
    for(i = 0; i < n; i++) t1[i] = t3[i];
    free(t3);
    return s1;
}
```

Figure 2: Sample implementation of memcpy.

```
void *memcpy(void *s1, const void *s2, size_t n);
    char *t1 = s1;
    const char *t2 = s2;
    while(n-- > 0) *t1++ = *t2++;
    return s1;
}
```

copies may safely be used for both loops. The implementation of `memcpy`, on the other hand, provides the compiler no basis for ruling out the possibility that, for example, `s1` and `s2` point to successive bytes. Therefore unconditional use of block copies does not appear to be safe, and the code generated for the single loop in `memcpy` may not be as fast as the code for the loops in `memmove`.

## 2.3 Restricted pointers

If an aliasing restriction like the one for `memcpy` could be expressed in a function definition, then it would be available to a compiler to facilitate effective pointer alias analysis. The preceding discussion suggests the form that the restriction should take. It should be possible to specify in the declaration of a pointer that it provides "exclusive initial access" to the object to which it points, as though the pointer were initialized with a call to `malloc`. Because it is the unqualified versions of function parameter types that are compared for type compatibility, it is convenient to specify the restriction with a type qualifier, spelled `restrict`, on the pointer type. The prototype for `memcpy` in Figure 3 thus both expresses the desired restriction and is compatible with the current prototype.

Figure 3: Restricted pointer prototype for memcpy.

```
void *memcpy(void *restrict s1, const void *restrict s2, size_t n);
```

## 2.4 Design choices

Given this general concept of restricted pointers, there remain a number of design choices. Three of these are discussed in detail in an appendix. Most such choices may be characterized as a choice between simplicity and expressive power. To some, it seems better to simplify the analysis of restricted pointers by confining their use to the simplest paradigms. For example, restricted pointers would be quite useful even if they could be declared only as function parameters and if the restricted pointers themselves could not be modified. This would support what is clearly the most important paradigm, an alias-free function call interface in C, analogous to that in Fortran. To others, it seems better to support the expression of aliasing restrictions in as many paradigms as feasible. This would be helpful in converting existing programs to use restricted pointers, and would allow more freedom of style in new programs.

The definition given in section 3 favors the second view. It allows restricted pointers to be modifiable, to be members of structures and elements of arrays, and to be "strongly scoped," in the sense that a restricted pointer declared in a nested block makes a non-aliasing assertion only within that block.

# 3 Formal definition of restrict

What follows are proposed additions to the document *International Standard Programming Languages — C (ISO/IEC 9899:1990(E))*.

## Keywords [ISO §6.1.1]

*Add a keyword:* **restrict**

## Type Qualifiers [ISO §6.5.3]

### Syntax

*Add a type-qualifier:* **restrict**

### Constraints

*Add a constraint:*

Types other than pointer types derived from object or incomplete types shall not be restrict-qualified.

### Semantics

*Add the following text after line 22.*

Let `D` be a declaration of an ordinary identifier that provides a means of designating an object `P` as a restrict-qualified pointer.

If `D` appears inside a block and does not have storage-class `extern`, let `B` denote the block. If `D` appears in the list of parameter declarations of a function definition, let `B` denote the associated block. Otherwise, let `B` denote the block of `main` (or the block of whatever function is called at program startup, in a freestanding environment).

In what follows, a pointer expression `E` is said to be *based* on object `P` if (at some sequence point in the execution of `B` prior to the evaluation of `E`) modifying `P` to point to a copy of the array object into which it formerly pointed would change the value of `E`. (In other words, `E` depends on the value of `P` itself rather than on the value of an object referenced indirectly through `P`. For example, if identifier `p` has type `(int **restrict)`, then the pointer expressions `p` and `p+1` are based on the restricted pointer object designated by `p`, but the pointer expressions `*p` and `p[1]` are not. Note that "based" is defined only for expressions with pointer types.)

During each execution of `B`, let `O` be the array object that is determined dynamically by all references through pointer expressions based on `P`. Then *all* references to values of `O` shall be through pointer expressions based on `P`. Furthermore, if `P` is assigned the value of a pointer expression `E` that is based on another restricted pointer object `P2`, associated with block `B2`, then either the execution of `B2` shall begin before the execution of `B`, or the execution of `B2` shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.

Here an execution of `B` means that portion of the execution of the program during which storage is guaranteed to be reserved for an instance of an object that is associated with `B` and has automatic storage duration. A reference to a value means either an access to or a modification of the value. During an execution of `B`, attention is confined to those references that are actually evaluated (this excludes references that appear in unevaluated expressions, and also excludes references that are "available," in the sense of employing visible identifiers, but do not actually appear in the text of `B`).

A translator is free to ignore any or all aliasing implications of uses of `restrict`.

# 4 Examples

## 4.1 File scope restricted pointers

A file scope restricted pointer is subject to very strong restrictions. It should point into a single array object for the duration of the program. That array object may not be referenced both through the restricted pointer and through either its declared name (if it has one) or another restricted pointer. Because of these restrictions, references through the pointer can be optimized as effectively as references to a file scope array through its declared name. File scope restricted pointers are therefore useful for providing access to dynamically allocated global arrays. In Figure 4, a compiler can deduce from the restrict qualifiers that there is no potential aliasing among the names `a`, `b`, and `c`. Notice how the single block of allocated storage is "subdivided" into two unique arrays in `init`.

Figure 4: File scope restricted pointer.

```
float *restrict a, *restrict b;
float c[100];

void init(int n) {
    float *t =  malloc(2 * n * sizeof(float));
    a = t;      /* a refers to 1st half. */
    b = t + n;  /* b refers to 2nd half. */
}
```

## 4.2 Function parameters

Restricted pointers are also very useful as pointer parameters of a function. In the function `f3` in Figure 5, it is possible for a compiler to infer that there is no aliasing of modified objects, and so to optimize the loop aggressively. Upon entry to `f3`, the restricted pointer `a` must provide exclusive access to its associated array. In particular, within `f3` neither `b` nor `c` may point into the array associated with `a`, because neither is assigned a pointer value based on `a`. For `b`, this is evident from the const-qualifier in its declaration, but for `c`, an inspection of the body of `f3` is required.

Figure 5: Restricted pointer function parameters.

```
float x[100];
float *c;

void f3(int n, float *restrict a, float *const b) {
    int i;
    for ( i = 0; i < n; i++ )
        a[i] = b[i] + c[i];
}
void g3(void) {
    float d[100], e[100];
    c = x; f3(100,   d,    e); /* Behavior defined.   */
           f3( 50,   d, d+50); /* Behavior defined.   */
           f3( 99, d+1,    d); /* Behavior undefined. */
    c = d; f3( 99, d+1,    e); /* Behavior undefined. */
           f3( 99,   e,  d+1); /* Behavior defined.   */
}
```

Two of the calls shown in `g3` result in aliasing that is inconsistent with the `restrict` qualifier, and their behavior is undefined. Note that it is permitted for `c` to point into the array associated with `b`. Note also that, for these purposes, the "array" associated with a particular pointer means only that portion of an array object which is actually referenced through that pointer.

For a similar example in which `c` is another parameter instead of a file scope array, see "Aliasing of unmodified objects" in the appendix on "Discussion of design choices."

## 4.3 Block scope

A block scope restricted pointer makes an aliasing assertion that is limited to its block. This seems more natural than allowing the assertion to have function scope. It allows local assertions that apply only to key loops, for example. It also allows equivalent assertions to be made when inlining a function by converting it into a macro. In Figure 6, the original restricted pointer parameter is represented by a block scope restricted pointer.

Figure 6: Macro version of f3.

```
float x[100];
float *c;

#define f3(N, A, B)                                     \
{   int n = (N);                                        \
    float *restrict a = (A);                            \
    float *const    b = (B);                            \
    int i;                                              \
    for ( i = 0; i < n; i++ )                           \
        a[i] = b[i] + c[i];                             \
}
```

## 4.4 Members of structures

A restricted pointer member of a structure makes an aliasing assertion, and the scope of that assertion is the scope of the ordinary identifier used to access the structure. Thus although the structure type is declared at file scope in Figure 7, the assertions made by the declarations of the parameters of `f4` have block (of the function) scope.

Figure 7: Restricted pointers as members of a structure.

```
struct t {      /* Restricted pointers assert that    */
    int n;      /* members point to disjoint storage. */
    float *restrict p;
    float *restrict q;
};

void f4(struct t r, struct t s) {
    /* r.p, r.q, s.p, s.q should all point to    */
    /* disjoint storage during each execution of f4. */
    /* ... */
}
```

## 4.5 Type definitions

A `restrict` qualifier in a type definition makes an aliasing assertion when the typedef name is used in the declaration of an ordinary identifier that provides access to an object. As with members of structures, it is the scope of the latter identifier, not the scope of the typedef name, that determines the scope of the aliasing assertion.

## 4.6 Expressions based on restricted pointers

In Figure 8, the restricted pointer parameter `p` is potentially adjusted to point into a copy of its original array of two structures. By definition, a subsequent pointer expression is said to be based on `p` if and only if its value is changed by this adjustment. In the comment, the values of the pointer expressions in the first column are changed by this adjustment, and so those expressions are based on `p`. The values of the pointer expressions in the second column are not changed by the adjustment, and so those expressions are not based on `p`. This can be verified by adding appropriate print statements for the expressions, and comparing the values produced by the two calls of `f5` in `main`.

It is important to note that the definition of "based on" applies to expressions that rely on implementation-defined behavior. This is illustrated in this example, which assumes that the casts `(int)` followed by `(struct t *)` give the original value.

Figure 8: Pointer expressions based on p.

```
#include <stdlib.h>
#include <string.h>

struct t { int *q; int i; } a[2] = { /* ... */ };

void f5(struct t *restrict p, int c)
{
    struct t *q;
    int n;
    if(c) {
        struct t *r;
        r = malloc(2*sizeof(*p));
        memcpy(r, p, 2*sizeof(*p));
        p = r;
    }
    q = p;
    n = (int)p;
    /* - - - - - - - - - - - - - - - - - - - - - - - -
      Pointer expressions     Pointer expressions
      based on p              not based on p

      p                       p -> q
      p + 1                   p[1] . q
      &p[1]                   &p
      &p[1] . i
      &p->q
      q                       q -> q
      ++q
      (char *)p               (char *)(p -> i)
      (struct t *)n           ((struct t *)n) -> q
    - - - - - - - - - - - - - - - - - - - - - - - - */
}

main() {
    f5(a, 0);
    f5(a, 1);
}
```

## 4.7 Assignments between restricted pointers

Let us say that one restricted pointer is "newer" than another if the block with which the first is associated begins execution after the block associated with the second. Then the formal definition allows a newer restricted pointer to be assigned a value based on an older restricted pointer. This allows, for example, a function with a restricted pointer parameter to be called with an argument that is a restricted pointer.

Conversely, an older restricted pointer may be assigned a value based on a newer restricted pointer only after execution of the block associated with the newer restricted pointer has ended. This allows, for example, a function to return the value of a restricted pointer that is local to the function, and the return value then to be assigned to another restricted pointer.

The behavior of a program is undefined if it contains an assignment between two restricted pointers that does not fall into one of these two categories. Some examples are given in Figure 9.

Figure 9: Assignments between restricted pointers.

```
int *restrict p1, *restrict p2;

void f6(int *restrict q1, int *restrict q2)
{
    p1 = p2;     /* Behavior undefined. */
    p1 = q1;     /* Behavior undefined. */
    q1 = q2;     /* Behavior undefined. */
    {
        int *restrict r1, *restrict r2;
        ...
        r1 = r2; /* Behavior undefined. */
        q1 = r1; /* Behavior undefined. */
        p1 = r1; /* Behavior undefined. */
        ...
    }
}
```

## 4.8 Assignments to unrestricted pointers

This proposal permits assignment of the value of a restricted pointer to an unrestricted pointer, as in Figure 10.

Figure 10: Assigning restricted to unrestricted pointers.

```
void f7(int n, float *restrict r, float *restrict s) {
    float *p = r, *q = s;
    while(n-- > 0)
        *p++ = *q++;
}
```

If a compiler tracks pointer values, it should be able to optimize the loop as effectively as if the restricted pointers `r` and `s` were used directly.

More complicated ways of combining restricted and unrestricted pointers are less likely to be effective because they are more difficult for a compiler to analyze. As always, a programmer concerned about performance must adapt his style to the capabilities of available compilers. A conservative approach would be to avoid using both restricted and unrestricted pointers in the same function.

## 4.9 Ineffective uses of type qualifiers

Except where specifically noted in the formal definition, the `restrict` qualifier behaves in the same way as `const` and `volatile`. In particular, note that it is not a constraint violation for a function return type or the *type-name* in a cast to be qualified, but the qualifier has no effect because function call and cast expressions are not lvalues. Thus the presence of the `restrict` qualifier in the declaration of `f8` in Figure 11 makes no assertion about aliasing in functions that call `f8`. Similarly, the two casts make no assertion about aliasing of the references through the pointers `p` and `r`.

Figure 11: Qualified function return type and casts.

```
float *restrict f8(void)  /* No assertion about aliasing. */
{
    extern int i, *p, *q, *r;

    r = (int *restrict)q; /* No assertion about aliasing. */

    for(i = 0; i < 100; i++)
        *(int *restrict)p++ =  r[i];   /* No assertion    */
                                       /* about aliasing. */
    return p;
}
```

## 4.10 Constraint violations

It is a violation of the constraint in ISO §6.5.3 (of the Standard as modified above) to restrict-qualify an object type which is not a pointer type, or to restrict-qualify a pointer to a function.

Figure 12: Restrict cannot qualify non-pointer object types.

```
    int restrict x;   /* Constraint violation. */
    int restrict *p;  /* Constraint violation. */
```

Figure 13: Restrict cannot qualify pointers to functions.

```
float (*restrict f9)(void); /* Constraint violation. */
```

# 5 Appendix: Overlapping objects

The restriction in the *Description* of `memcpy` prohibits copying between "overlapping objects." An interpretation from X3J11 and WG14 in response to DR #042, X3J11 92-001, clarified what is meant by "objects" in this context. The following quotation from that defect report provides a background for the notion of an "object that is determined dynamically" that appears in the formal definition of `restrict`.

> From subclause 3.14, an object is "a region of data storage ... Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined ..."
>
> Figure 14: Memcpy between rows of a matrix.
>
> ```
> void f1(void) {
>     extern char a[2][N];
>     memcpy(a[1], a[0], N);
> }
> ```
>
> Therefore, the "objects" referred to by subclause 7.11.2.1 are exactly the regions of data storage pointed to by the pointers and dynamically determined to be of `N` bytes in length (i.e. treated as an array of `N` elements of character type).
>
> a) So, no, the objects are not "the largest objects into which the arguments can be construed as pointing".
>
> b) In Figure 14, the call to `memcpy` has defined behavior.
>
> c) The behavior is defined because the pointers point into different (non-overlapping) objects.
>
> Figure 15: Memcpy between halves of an array.
>
> ```
> void f2(void) {
>     extern char b[2*N];
>     memcpy(b+N, b, N);
> }
> ```
>
> a) Yes, for `memcpy`, a contiguous sequence of elements within an array can be regarded as an object in its own right.
>
> b) The objects are not the smallest contiguous sequence of bytes that can be construed; they are exactly the regions of data storage starting at the pointer and of `N` bytes in length.
>
> c) Yes, the non-overlapping halves of array `b` can be regarded as objects in their own rights.
>
> d) The behavior (in Figure 15) is defined.
>
> ...
>
> Length is determined by "various methods." For strings in which all elements are accessed, length is inferred by null byte termination. For `mbstowcs`, `wcstombs`, `strftime`, `vsprintf`, `sscanf`, `sprintf`, and all other similar functions, it was the intent of the standard that the rules in subclause 7.11.1 be applicable by extension (i.e., the objects and lengths are similarly dynamically determined).

# 6 Appendix: Discussion of design choices

## 6.1 Contiguity

A restricted pointer points to an associated object that is determined dynamically. When all references, both direct and indirect, through a restricted pointer comprise a contiguous sequence of bytes, then the associated object is exactly that sequence. Otherwise, the object consists of both the referenced bytes and those unreferenced bytes that fall between any two referenced bytes.

This convention is consistent with the specification noted above, that "except for bit-fields, objects are composed of contiguous sequences of one or more bytes." It is also consistent with the notions of objects and argument association in Fortran 77. Thus, in environments that support mixed-language programming, restricted pointer parameters can be used in C prototypes for functions defined in Fortran, and also in definitions of C functions that are called from Fortran.

Note that an implementation is free to support restricted pointers to non-contiguous sets of bytes as an extension. Such support is not mandated, however, because the need for it does not seem compelling, and it could inhibit optimizations for architectures that cannot load and store single bytes or that have software-managed caches. Note that such an extension might be useful in implementation-defined calling conventions between C and Fortran 90, because Fortran 90 permits arguments that are non-contiguous sets of array elements.

## 6.2 Aliasing of unmodified objects

Comparison with Fortran raises an issue concerning aliasing of unmodified objects, as illustrated in Figure 16. Knowing that parameter `a` provides exclusive access to the array into which it points is sufficient to allow the loop in `f10` to be vectorized or executed in parallel. Knowing that `b` and `c` are not used to access overlapping portions of a single array allows no additional optimizations, and so, in principle, there is no need to restrict-qualify `b` and `c`.

Figure 16: Aliasing matters only for modified objects.

```
void f10(int n, float *restrict a, float *b, float *c) {
    int i;
    for ( i = 0; i < n; i++ )
        a[i] = b[i] + c[i];
}
```

But, as a practical matter, it is easier for a compiler to analyze functions in which all pointer parameters are restrict-qualified. This suggests that the semantics of the qualifier should be defined so that one can restrict-qualify all three pointer parameters of `f10` and still make calls of the form `f10(x,y,y)`. This would be analogous to the aliasing semantics of Fortran dummy arguments.

The simplest way of doing this (proposed in Draft 1 of this report) is to allow aliasing through two restrict-qualified pointers provided the referenced objects are not modified. Unfortunately, if those objects are themselves pointers (i.e., there are two levels of indirection), this aliasing can inhibit optimization, even if the secondary pointers are also restrict-qualified and used to modify the objects to which they, in turn, refer. See X3J11.1/93-040 for examples.

In the final analysis, it did not seem possible to permit aliasing of unmodified objects, have effective assertions for multiple levels of indirection, and still keep the semantics relatively simple.

It was decided to drop the special treatment of unmodified objects, largely because the practical effect is quite small. First, as noted, the extra qualifiers are not needed in principle, though they may be useful for specific compilers. Second, in these cases, it is also useful, and might be sufficient, to const-qualify `b` and `c`, because this gives a compiler enough aliasing information in the parameter declarations alone to optimize `f10` (see the discussion of Figure 5). Finally, even if a specific compiler requires all three restrict qualifiers to give the desired optimization, it is most unlikely that their use would give unexpected results for a call such as `f10(x,y,y)`, even though its behavior is technically undefined. Such use may therefore be justified in particular cases, as are other instances of undefined or implementation-defined behavior, such as assigning to one member of a union and then referring to another.

## 6.3 Linked lists

Operations on data contained in the items of a linked list are difficult to optimize, particularly for vector or parallel execution. Because potential aliasing is not the only difficulty, it is unlikely that the use of restricted pointers in this context will make much difference with currently available compilers. Nevertheless, as the use of parallel architectures becomes more widespread, their compilers may well be more ambitious in this area. With this in mind, the examples in Figure 17 illustrate how to use the `restrict` and `const` qualifiers to make assertions that could, in principle, enable effective optimizations.

First note that restricted pointers cannot, in general, be used for link pointers in portions of the program that modify the lists. Thus in Figure 17, the assignments to `next` and `head` would give undefined behavior if they were restrict qualified.

Figure 17: Operations on the structure of a linked list.

```
struct t { struct t *next; float x; } *head, *item;

void modify_links()
    /* Add a list item at the beginning. */
    item -> next = head; head = item;

    /* Add a list item as the second item. */
    item -> next = head -> next; head -> next = item;

    /* Delete second list item. */
    head -> next = head -> next -> next;

    /* Delete first list item. */
    head = head -> next;
}
```

In contrast, in functions in which the structure of the lists is constant, this can be asserted by using a combination of the `restrict` and `const` qualifiers. Note that, as in Figure 18, this requires a second structure type, and a cast from the original to the new type. The comments describe an optimization that allows parallel execution of the second loop. Alternatively, the second loop could be fused with the first, so that both operations are performed during a single walk. The latter optimization is likely to be beneficial on any architecture.

Unfortunately, there is a subtle problem with this approach. If there is a function call between the two loops that has `rhead` as a parameter, then there should be a guarantee that the cached links are not changed during the execution of the called function. The problem is that there is nothing to prevent the function from modifying a link via

```
*(struct rt *restrict *)&(rhead -> next) = ...
```

(assuming for the sake of argument that the structures on the list were dynamically allocated as an array of structures). To make the aliasing assertions effective would require forbidding this sort of modification. This might be done be extending the full semantics of the `const` qualfier to include objects designated as const-qualified by a restricted pointer, as well as objects that are const-qualified by their definitions. (This extension is not part of the current proposal.)

Figure 18: Operations on the values in a linked lists.

```
struct t { struct t *next; float x; };
struct rt { struct rt *const restrict next; float x; };

void modify_values(struct t *head) {
    /* Assert that links will not change. */
    struct rt *const restrict rhead = (struct rt *)head;

    /* Need modifiable unrestricted pointer to walk list. */
    struct rt *this;

    /* Can build a cache of link addresses. */
    for(this = rhead; this; this = this -> next) {
        this -> x += 1;
    }

    /* Can use cached addresses to execute in parallel. */
    for(this = rhead; this; this = this -> next) {
        this -> x *= 2;
    }
}
```

# 7 Appendix: Array syntax and qualified parameters

By ISO §6.7.1, a declaration of a function parameter as "array of *type*" is adjusted to "pointer to *type*." Thus the two prototypes in Figure 19 for `f11` are compatible.

Figure 19: Equivalent parameter declarations.

```
void f11(int n, float a[][100], float b[][100]);
void f11(int n, float (*a)[100], float (*b)[100]);
```

In either case, `a` and `b` are pointers used to access their respective two-dimensional arrays, but the first form more clearly conveys the rank of the arrays. It is therefore unfortunate that only the second form allows the pointers to be qualified, e.g., by `restrict`.

This suggests an extension of the syntax to allow one or more type qualifiers to appear within the "array of" type derivation for a function parameter declared to have array type. The declaration is then adjusted to the analogously qualified pointer type.

Figure 20: Restricted version of f11.

```
void f11(int n, float a[restrict][100], float b[restrict][100])
{
    int i, j;
    for(i = 0; i < n; i++)
        for(j = 0; j < 100; j++)
           a[i][j] = b[i][j];
}

void g11(int m, int k, float p[][100]) {
    f11(m, p, p+k); /* Defined if and only if m <= k. */
}
```

This extension would be even more convenient in the presence of a variable length array extension, that would allow another parameter to be used in an array size expression. In Figure 21, the compiler, or code reviewer, can use the information in the prototype alone to deduce that the first call of `f12` in `g12` is defined, but the second call is potentially undefined (depending upon which elements of the arrays `f12` actually references).

Figure 21: Restrict and variable length arrays.

```
void f12(int m, int n, float a[restrict m][n],
                       float b[restrict m][n]);

void g12(int n, float p[][n]) {
        f12(10, n, p, p+10); /* Defined behavior. */
        f12(20, n, p, p+10); /* Potential undefined behavior. */
}
```

For consistency, the extension would apply to all type qualifiers. The following changes to the Standard would be required for this extension.

## Declarators [ISO §6.5.4]

### Syntax

*Allow an optional* type-qualifier-list *in the third form of direct-declarator:*
*direct-declarator* `[` *type-qualifier-list* $_{opt}$ *constant-expression* $_{opt}$ `]`

## Array Declarators [ISO §6.5.4.2]

### Constraints

*Add a constraint:*

Type qualifiers shall appear preceding or in place of a size expression only in a declaration of a function parameter of array type, and then only in the outermost array type derivation.

## Function Definition [ISO §6.7.1]

### Semantics

*Modify lines 23-24 to read:*

...A declaration of a parameter as "array of *type*" shall be adjusted to "qualified pointer to *type*" (where the type qualifiers are those specified within the square brackets of the array type derivation), ...

# 8 Appendix: Comparison with noalias

The following is an overview of the differences between `restrict` and the previously proposed `noalias`.

The X3J11 committee attempted to solve the aliasing problem in C by introducing a new type qualifier `noalias`. That effort failed because of technical problems with the proposed semantics of `noalias`. This restricted pointer proposal is different in many ways.

- Only pointers can be declared to be restricted. In the `noalias` proposal, all objects were permitted to be declared noalias-qualified.
- It is the declaration of the restricted pointer that makes the aliasing assertions, while it was the noalias-qualified lvalue that made the aliasing assertions.
- A restricted pointer can be an alias with an unrestricted pointer, whereas a pointer to a noalias-qualified type was guaranteed to be completely alias free.
- The proposed semantics of `noalias` defined an alternate execution path in which virtual objects were created and later synchronized with the original object. No alternate execution path is defined for restricted pointers. The proposed semantics for restricted pointers merely permit the optimizer to statically analyze restricted pointers.
- A block scope restricted pointer only makes assertions on the containing scope. In the `noalias` proposal, a block scope noalias-qualified object made assertions that affected the entire containing function.
- A pointer to a noalias-qualified type made no aliasing assertion if it was a member of a structure (because identifiers designating members of structures were not "handles"). A restricted pointer member of a structure does make an aliasing assertion.