

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.7.3.1 Formal definition of restrict

restrict
formal definition

Let **D** be a declaration of an ordinary identifier that provides a means of designating an object **P** as a restrict-qualified pointer to type **T**. 1502

Commentary

A typedef name, function, and enumeration constant are also ordinary identifiers. However, they do not declare objects.

C++

Support for the **restrict** qualifier is new in C99 and is not available in C++.

Example

```

1  #include <stdlib.h>
2
3  typedef int T;
4
5  T * restrict D;
6
7  void f(void)
8  {
9  D = (T * restrict)malloc(sizeof(T)); /* Allocate object P. */
10 }
```

restrict pointer
lifetime

If **D** appears inside a block and does not have storage class **extern**, let **B** denote the block. 1503

Commentary

This and the following two sentences define a region of program text, denoted by **B** (and **B2** elsewhere). This formal definition of **restrict** refers to events that occur before, during, or after the execution of this block.

Example

```

1  typedef int T;
2
3  void f(void)
4  {
5  T * restrict D /* Started D's associated block B. */
6                /* D is now visible in the block B. */
7                ;
8                /* Ended D's associated block B. */
```

If **D** appears in the list of parameter declarations of a function definition, let **B** denote the associated block. 1504

Commentary

```

1  typedef int T;
2
3  void f(T * restrict D /* D is now visible, outside of its associated block B. */
4                )
5  {
6  /* ... */
7  /* Ended D's associated block B. */
```

1505 Otherwise, let **B** denote the block of `main` (or the block of whatever function is called at program startup in a freestanding environment).

Commentary

The cases covered by this *otherwise* include all objects having static storage duration and file scope, and all objects having allocated storage duration.

1506 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**.¹¹⁷⁾

restrict
pointer based on

Commentary

This defines the term *based*. The use of the term *array object* here refers to a prior specification about a pointer to a non-array object behaving like a pointer to an array containing a single element (although technically that wording limits itself to “for the purposes of these operators”).

additive
operators
pointer to object

That is, the value of **E** depends on the value of **P**, not the value pointed to by **P**. It is possible that the former value of **P** is indeterminate, so the involvement of copies of formerly pointed-to array-objects is a conceptual one.

Example

```

1  #include <stdint.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  struct t {
6      int *q;
7      int i;
8      } a[2] = { 0, /* ... */ };
9
10 void WG14_N448(struct t *restrict P,
11               _Bool c_flag)
12 {
13     struct t *q;
14     intptr_t n;
15     /*
16      * Adjust P to point at copy of original object?
17      */
18     if (c_flag)
19     {
20         struct t *r;
21         r = malloc(2*sizeof(*P));
22         memcpy(r, P, 2*sizeof(*P));
23         P = r;
24     }
25     q = P;
26     n = (intptr_t)P;
27     /*
28      * Lists of pointer expressions that are, and are not,
29      * based on object P, in the execution of block B.
30      *
31      *      expression E      expression E
32      *      based on object P  not based on object P
33      *
34      *      P                  P->q
35      *      P+1                P[1].q
36      *      &P[1]               &P
37      *      &P[1].i

```

```

38 *      &P->q
39 *      q                q->q
40 *      ++q
41 *      (char *)P        (char *) (P->i)
42 *      (struct t *)n    ((struct t *)n)->q
43 */
44 }                        /* Block B has ended here. */
45
46 void f(void)
47 {
48     WG14_N448(a, 0);
49     WG14_N448(a, 1);
50 }

```

Note that “based” is defined only for expressions with pointer types.

1507

Commentary

For instance, in the example function, WG14_N448, given in the Example subclause of the previous C sentence, *based* is defined for $q \rightarrow q$, but not for $q \rightarrow i$.

During each execution of **B**, let **L** be any lvalue that has **&L** based on **P**.

1508

Commentary

This defines the term **L**. Starting with the address of **L**, that is based on **P**, we can use it to walk back through any chain of pointers.

Example

In the following we have $\&(p \rightarrow v[i])$ (equivalent to $(p \rightarrow v) + i$) based on the restricted pointer $p \rightarrow v$. Walking up the pointer chain, $\&(p \rightarrow v)$ is based on the restricted pointer p ,

```

1  typedef struct {
2      int n;
3      double * restrict v;
4      } vector;
5
6  void f(vector * restrict p)
7  {
8      p->v[i];
9  }

```

If **L** is used to access the value of the object **X** that it designates, and **X** is also modified (by any means), then the following requirements apply: **T** shall not be const-qualified.

1509

Commentary

This requirement means that, if **T** is const-qualified then this qualification cannot be cast away and the object **X** modified via some non-const-qualified lvalue **L** (if this sequence of operations occurred the behavior would be undefined).

Example

The use of **const** (via the typedef name **T**) and **restrict** in the definition of WG14_N866_F implies that **g** only accesses the value of $*p$, and does not modify it. On some processor architectures a translator could use this information to generate code to initiate the load from $*p$ before the call to **g**, making the value available sooner both within execution of **g** and for the **return** statement. The size of the performance benefit will depend on the size of the type **T**.

```

1  typedef const int T; /* Some type T. */
2  extern void g();
3
4  T WG14_N866_F(T * restrict p)
5  {
6  g(p);
7  return *p;
8  }

```

1510 Every other lvalue used to access the value of **X** shall also have its address based on **P**.

Commentary

Automatically deducing, during program translation, the set of lvalues used to access **X** is potentially very expensive, computationally. Being able to assume that such lvalues are based on **P** (otherwise the behavior is undefined) considerably reduces an optimizer's computational overhead (when performing optimizations that rely on points-to information).

1511 Every access that modifies **X** shall be considered also to modify **P**, for the purposes of this subclause.

Commentary

There are a number of ways wording in the standard can promote optimization. Possibilities include: (1) bounding the behavior for common cases (such as limits on how objects referenced by restricted pointers are accessed), and (2) ensuring that edge cases have undefined behavior (a translator does not have to bother taking them into account, the behavior can be whatever happens to occur). The purpose of this C sentence is to render some awkward edge cases undefined behavior.

Example

When the one or more parameters of a function definition have a restricted pointer type a translator needs to be able to optimize the function body, independent of the actual arguments passed. Passing the address of an object as the value of two separate arguments, in a function call, occurs reasonably frequently. The following examples are based on a discussion in document WG14/N866 written by Homer.

¹⁵¹⁹ [restrict example 2](#)

```

1  typedef struct {
2      int n;
3      double * restrict v;
4      } vector;
5
6  void WG14_N866_C(vector * restrict p, vector * restrict q)
7  {
8  int i;
9  for(i=1; i < p->n; i++)
10     p->v[i] += q->v[i-1];
11 }

```

What behavior does this C clause give to the calls `WG14_N866_C(&x, &y)` and `WG14_N866_C(&x, &x)` (when `x.n > 1`)?

1. **B** is the block formed by the body of the definition of `WG14_N866_C`,
2. Let **L** be the lvalue `p->v[i]`, and **X** the object which it designates,
3. Let **X** be `p->v[i]`,
4. `&(p->v[i])` (equivalent to `(p->v)+i`) is based on the restricted pointer `p->v`,
5. a modification of **X** is considered also to modify the object designated by `p->v` (current C sentence),

¹⁵⁰⁸ [restrict &L based on L](#)

6. recursively, $\&(p \rightarrow v)$ is based on the restricted pointer p ,
7. when the arguments passed are $(\&x, \&y)$ p and q refer to different objects, and no other pointer is based on p ,
8. when the arguments are $(\&x, \&x)$ the same object is also accessed through the lvalue $q \rightarrow v$, but $\&(q \rightarrow v)$ is not based on p and therefore behavior is undefined.

restrict 1513
undefined behavior

A translator is thus able to unconditionally optimize the body of the function `WG14_N866_C` (either the **restrict** semantics does hold, or the undefined behavior can be that it does :-).

In the following function definition, even if $*a$ and $*b$ happen to refer to the same object, the members $a \rightarrow p$ and $b \rightarrow q$ will be distinct restricted pointers. A translator can infer that $*(a \rightarrow p)$ and $*(b \rightarrow q)$ are distinct objects, and so there is no potential aliasing to inhibit optimization of the body of f .

```

1  typedef struct {
2      int * restrict p;
3      int * restrict q;
4      } T;
5
6  int WG14_N866_E(T * restrict a, T * restrict b)
7  {
8      *(a->p) += 1;
9      return *(b->q);
10 }

```

During an execution of f , the only object that is actually modified is $(*a) \rightarrow p$. Now $\&((*a) \rightarrow p)$ is $a \rightarrow p$, a restricted pointer, so $a \rightarrow p$ is also considered to be modified. Recursively, $\&(a \rightarrow p)$ is based on a , another restricted pointer. Because no other lvalues are used to access either $(*a) \rightarrow p$ or $a \rightarrow p$, all the requirements of the specification are met (and they do not involve b).

restrict B2

If P is assigned the value of a pointer expression E that is based on another restricted pointer object P_2 , associated with block B_2 , then either the execution of B_2 shall begin before the execution of B , or the execution of B_2 shall end prior to the assignment.

1512

Commentary

This requirement allows restricted pointers to be passed as arguments to function calls, and a restricted pointer within a nested scope to be assigned the value of a restricted pointer from an outer scope. If the value of a restricted pointer is assigned to another restricted pointer, e.g., P_{nest} , in a nested scope, the requirement applies only within the scope of P_{nest} and only to objects modified within that scope that are referenced through expressions based on P_{nest} at least once.

The case where execution of B_2 ends prior to the assignment covers the situation where a value, based on the restricted pointer object P_2 , returned from a function call is assigned to P .

restrict 1521
example 4

Coding Guidelines

Parallels can be drawn between the coding guideline discussion on the assignment of object addresses to pointers and the assignment of restricted pointers to other restricted pointer objects.

pointer indeterminate

Example

```

1  #define INLINE(P_1, P_2, N) {
2      int * restrict q_1 = P_1;
3      int * restrict q_2 = P_2;
4      for (int i = 0; i < N; i++)
5          {
6              q_1[i] += q_2[i];
7          }
8  }

```

1513 If these requirements are not met, then the behavior is undefined.

Commentary

In particular, an implementation is free to make the general assumption that the requirements are met and to perform the optimizations that are performed when they are met.

restrict
undefined
behavior

1514 Here an execution of **B** means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration associated with **B**.

Commentary

This sentence clarifies the meaning by using an existing, well defined, term, lifetime.

lifetime
of object

restrict
execution of
block means

1515 A translator is free to ignore any or all aliasing implications of uses of **restrict**.

Commentary

A developer may be able to deduce translator behavior from the performance of the program image, or by examining the undefined behavior that occurs for certain constructs.

1513 restrict
undefined be-
havior

Coding Guidelines

Uses of **restrict** provide information on source code properties believed to be true by developers. Something that static analysis tools can check and make use of.^[1]

1516 EXAMPLE 1 The file scope declarations

```
int * restrict a;
int * restrict b;
extern int c[];
```

assert that if an object is accessed using one of **a**, **b**, or **c**, and that object is modified anywhere in the program, then it is never accessed using either of the other two.

Commentary

The declaration of **c** does not explicitly assert anything about **restrict** related semantics. It just so happens that all of the other declarations are restrict-qualified, which implies additional semantics on accesses to **c**. If another declaration, say `extern int *d;`, appeared in the list, then the same object could be accessed using either **c** or **d**, but the stated relationship between **a**, **b**, and **c** would remain true (and so would an equivalent one between **a**, **b**, and **d**).

1517 117) In other words, **E** depends on the value of **P** itself rather than on the value of an object referenced indirectly through **P**.

Commentary

The specification of **restrict** provides a mechanism for developers to give guarantees about the set of objects a pointer may point at. Translators may use these guarantees to deduce information about the value of an object referenced indirectly though so qualified pointers.

footnote
117

1518 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.

Commentary

If **p** had type (`int * restrict * restrict`) then both ***p** and **p[1]** would be based on the restricted pointer ***p**.

restrict
example 2

EXAMPLE 2 The function parameter declarations in the following example

1519

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

assert that, during each execution of the function, if an object is accessed through one of the pointer parameters, then it is not also accessed through the other.

The benefit of the **restrict** qualifiers is that they enable a translator to make an effective dependence analysis of function **f** without examining any of the calls of **f** in the program. The cost is that the programmer has to examine all of those calls to ensure that none give undefined behavior. For example, the second call of **f** in **g** has undefined behavior because each of **d[1]** through **d[49]** is accessed through both **p** and **q**.

```
void g(void)
{
    extern int d[100];
    f(50, d + 50, d); // valid
    f(50, d + 1, d); // undefined behavior
}
```

Commentary

As this example states, the burden of correctness lies with the programmer. A translator is entitled to act as if what the programmer said (through the use of **restrict**) is correct.

Other Languages

Some languages (e.g., Fortran and Ada) copying operations such as these can be performed through the use of array slicing operators.

Coding Guidelines

The technical difficulties involved in proving that a developer's use of **restrict** has defined behavior are discussed elsewhere.

alias analysis

EXAMPLE 3 The function parameter declarations

1520

```
void h(int n, int * restrict p, int * restrict q, int * restrict r)
{
    int i;
    for (i = 0; i < n; i++)
        p[i] = q[i] + r[i];
}
```

illustrate how an unmodified object can be aliased through two restricted pointers. In particular, if **a** and **b** are disjoint arrays, a call of the form **h(100, a, b, b)** has defined behavior, because array **b** is not modified within function **h**.

Commentary

Information on when objects may, or may not, be modified is of primary importance to the optimization process. Lack of information on when two read accesses refer to the same object may result in suboptimal code, but it does not prevent some optimizations being made. The committee did not consider it worth addressing this “reads via two apparently independent pointers” issue in C99.

restrict
example 4

EXAMPLE 4 The rule limiting assignments between restricted pointers does not distinguish between a function call and an equivalent nested block. With one exception, only “outer-to-inner” assignments between restricted pointers declared in nested blocks have defined behavior.

1521

```
{
    int * restrict p1;
    int * restrict q1;
    p1 = q1; // undefined behavior
    {
        int * restrict p2 = p1; // valid
        int * restrict q2 = q1; // valid
        p1 = q2;                // undefined behavior
        p2 = q2;                // undefined behavior
    }
}
```

The one exception allows the value of a restricted pointer to be carried out of the block in which it (or, more precisely, the ordinary identifier used to designate it) is declared when that block finishes execution. For example, this permits `new_vector` to return a `vector`.

```
typedef struct { int n; float * restrict v; } vector;
vector new_vector(int n)
{
    vector t;
    t.n = n;
    t.v = malloc(n * sizeof (float));
    return t;
}
```

Commentary

These exceptions are discussed elsewhere.

1512 [restrict](#)
B2

References

1. A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *SIGPLAN Conference on Programming Language Design and Implementation PLDI'03*, pages 129–140, June 2003.