Clarifying the restrict Keyword v2

Doc. No.: WG14/N2260
Date: 2018-05-20
Author 1: Troy A. Johnson
Email: troyj@cray.com
Author 2: Bill Homer
Email: homer@cray.com

Introduction

Drafts of the proposal to add the restrict qualifier to the C standard contained more examples than were eventually included in the standard. The examples that were included were limited to cases in which all pointers were restricted, which are most helpful to a translator. The examples that were not included served to show what the formal definition implied in cases in which not all pointers were restricted.

The absence of those additional examples may have been intended to avoid encouraging usages that are needlessly challenging for a translator, but they have also apparently lead to confusion about how to interpret the formal definition in such cases. That confusion has resulted in translators that infer non-aliasing in cases where the standard does not imply it, and so make unsafe optimizations.

This proposal seeks to clarify the restrict keyword, both for programmers and for implementers of translators, by adding examples in which only a single pointer is restrict-qualified. They illustrate that the formal definition can effectively imply non-aliasing in such cases, but only with a careful analysis that considerably exceeds what is necessary for the examples currently in the standard. Accompanying text makes it clear that a translator may reasonably decline to attempt such analysis, and so programs meant to be portable should avoid relying on such usages.

Problem

Consider these functions:

```
    void f1(int *p, int *q) { ... }
    void f2(int * restrict p, int * restrict q) { ... }
    void f3(int * restrict p, int *q) { ... }
    void f4(int *p, int * restrict q) { ... }
```

In f1, a translator must assume that p and q may point to the the same object, or to different elements of the same array. In f2, the use of restrict guarantees that such aliasing does not happen, at least for objects that are modified during the execution of f2.

Although there is universal agreement on the difference between f1 and f2, there is disagreement among programmers and implementations as to whether f2, f3, and f4 are semantically equivalent. Note that the examples provided in the standard do not help resolve this disagreement, since all examples are similar to f2.

A careful reading of C11 6.7.3 leads to the conclusion that f2, f3, and f4 do NOT provide identical information. An implementation is able to infer without examining the body of f2 that accesses through p and q do not alias. Both f3 and f4 admit the possibility that the unrestricted pointer becomes "based" (see 6.7.3.1 paragraph 3) on the restricted pointer during execution of the function's body. For f3 this might happen directly with q = p and for f4 this might happen directly with p = q, but it could happen through less obvious, indirect means. This situation is impossible in f2 due to the limitations that 6.7.3.1 paragraph 4 places on assignments between two restricted pointers, but there is no such limitation on assigning a restricted pointer to an unrestricted pointer. Once the unrestricted pointer becomes based on the restricted pointer, it may be used to access the same object; such an access does not violate the standard because all accesses to the object are still being performed using a pointer that is based on the restricted pointer. Therefore, in order to have the same freedom to optimize f3 or f4 as f2, an implementation must analyze the function's body and prove that this situation does not occur. One generally should expect less optimization of f3 and f4 compared to f2, unless using a particular implementation that performs the necessary analysis, the code is easily analyzable, and the situation does not occur. An implementation reasonably might not attempt this analysis for a variety of reasons: trust that the programmer would have written f2 if their code permitted the second restrict, belief that f3 and f4 are too rare to be worth special effort, or belief that the analysis often will need to reach a conservative conclusion.

For a concrete example, try compiling the program below with and without optimization, but leave inlining disabled. The opaque function call to g() is used to represent an unanalyzable basing of q upon p.

```
    #include <stdio.h>

    void g(int **a, int *b)
    {
        *a = b;
    }

    int foo(int * restrict p, int *q)
    {
        g(&q, p); // effectively q = p
        *p = 1;
        *q = 2;
        return *p + *q;
    }

    int main()
    {
        int x, y;
        printf("%d\n", foo(&x, &y));
        return 0;
    }
```

Some compilers, like Clang 6.0, behave correctly:

```
    > clang -O0 test.c
    > ./a.out
```

```
4
> clang -O2 -fno-inline test.c
> ./a.out
4
```

Other compilers, like GNU 7.3.0, behave incorrectly:

```
> gcc -O0 test.c
> ./a.out
4
> gcc -O2 -fno-inline test.c
> ./a.out
3
```

One commercial compiler was found to display the incorrect behavior as well.

For an example where q becomes based on p in the same function, consider:

```
#include <stdio.h>

int z;

int foo(int * restrict p, int *q)
{
    if (z) q = p;  // Together, these two statements are q = p.  The
    if (!z) q = p; // conditions are used to prevent forwarding.
    *p = 1;
    *q = 2;
    return *p + *q;
}

int main()
{
    int x, y;
    printf("%d\n", foo(&x, &y));
    return 0;
}
```

For that program, the same commercial compiler continues to translate foo() as returning 3 (it folds the addition at compile time), but the GNU and Clang compilers both produce correct code.

It is our experience that this problem has gone unnoticed for so long because many programmers expect equivalent optimization for f2, f3, and f4. Such programmers fall into two groups. The first group does not know about the semantic difference among f2, f3, and f4. Because they think the implementation is being presented with identical information in all cases, they expect the same optimization behavior in all cases. The second group understands the semantic difference, but believes that a quality implementation contains analysis sufficient to optimize f3 or f4 like f2. Their reason for not including the second restrict is brevity. Both groups of programmers see their expectations met by implementations that are actually behaving incorrectly. When they encounter an implementation that behaves correctly, they conclude that the conforming implementation is the defective one!

Proposed Additions to C

The standard needs examples that clarify the meaning of a solitary restricted pointer so that programmers and language implementers can agree.

6.7.3.1

...

8. EXAMPLE 2 The function parameter declarations in the following example

```
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.
[ADD] The translator can make this no-aliasing inference based on the parameter declarations alone, without analyzing the function body.

[ADD] 13. EXAMPLE 5 Suppose that a programmer knows that references of the form p[i] and q[j] are never aliases in the body of a function:

```
void f(int n, int *p, int *q) { ... }
```

There are several ways that this information could be conveyed to a translator, using the restrict qualifier. Example 2 shows the most effective way, qualifying all pointer parameters, and can be used provided that neither p nor q becomes based on the other in the function body.

A potentially effective alternative is:

```
void f(int n, int * restrict p, int * const q) { ... }
```

Again it is possible for a translator to make the inference based on the parameter declarations alone, though now it must use subtler reasoning: that the const-qualification of q precludes it becoming based on p. There is also a requirement that q is not modified, so this way cannot be used for the function in Example 2, as written.

[ADD] 14. EXAMPLE 6 Another potentially effective alternative is:

```
void f(int n, int *p, int const * restrict q) { ... }
```

Again it is possible for a translator to make the inference based on the parameter declarations alone, though now it must use even subtler reasoning: that this combination of restrict and const means that objects referenced using q cannot be modified, and so no modified object can be referenced using both p and q.

[ADD] 15. EXAMPLE 7 The least effective alternative is:

```
void f(int n, int * restrict p, int *q) { ... }
```

Here the translator can make the inference only by analyzing the body of the function and proving that q cannot become based on p. Some translator designs may choose to exclude this analysis, given availability of the more effective alternatives above. Such a translator shall assume that aliases are present; assuming that aliases are not present may result in an incorrect translation. Also, a translator that attempts the analysis may not succeed in all cases and need to conservatively assume that aliases are present.