Proposal for C2y

WG14 N3659

Title: Considering expressions based on `restrict` pointers as pure rvalue expressions
Author, affiliation: Vladislav Belov, Syntacore
Date: 2025-07-25
Proposal category: Defect reports / Clarifications
Target audience: Developers working on C code bases and compiler developers

Abstract: The formal definition of `restrict` in the C standard introduces the concept of "an expression `E` based on the `restrict` pointer `P`". However, this definition remains somewhat ambiguous and can be interpreted in ways that allow other lvalue pointers, when assigned `E`, to be considered expressions based on `P` as well. This ambiguity is not only confusing for developers but also complicates compiler optimizations related to `restrict` pointers, which contradicts the primary purpose of introducing the `restrict` qualifier — enabling more effective optimizations.

Prior art: C

Considering expressions based on `restrict` pointers as pure rvalue expressions

Reply-to: Vladislav Belov (vladislav.belov@syntacore.com)
Document No: n3659
Date: 2025-07-25

Introduction and Rationale

Since the introduction of the `restrict` qualifier in C99, programmers are able to declare certain pointers in their programs with the `restrict` qualifier. In general, this qualifier serves as a hint to the compiler that some pointers do not alias each other within a certain region of code, which enables better optimization opportunities.

According to the standard, expressions involving arithmetic on a `restrict` pointer are considered to be "based on" that pointer. This behavior extends further, as highlighted by the informative note in the C standard (J.5.6, EXAMPLE 7), which states:

> The least effective alternative is:
>
> ```
> void f(int n, int * restrict p, int *q) { /* ... */ }
> ```
>
> Here the translator can make the no-aliasing 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 the availability of the more effective alternatives described previously. Such a translator is required to assume that aliases are present because 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 consequently needs to conservatively assume that aliases are present.

This example suggests that even simple copies of a `restrict`-qualified pointer into another pointer implicitly make the new pointer "based on" the original.

Example 4: Copying a `restrict` pointer — the new pointer is based on the original

```
int * restrict x;
int * y = x;  // y is now based on x
```

This assumption can be a serious problem blocking the optimization produced with a translator, since it will need to prove that any other pointer introduced in the same scope in which the `restrict` pointer was introduced is not "based on" it, which, in general, is more of a complication for the alias analysis of the compiler than a simplification for optimizations (which is what `restrict` was designed for).Examples 5 and 6 from Section 6.7.4.2 of the standard show an alternative way to avoid such problems with `restrict` pointers used as function arguments, but this approach is still inconvenient for local `restrict` pointers, `restrict` pointers as structure fields, and similar cases. Also, we have observed a behavior that may indicate a possible language defect. The contradiction is as follows: the `restrict` keyword is a type qualifier for pointer types, meaning it gives a corresponding pointer certain additional properties, as described in Section 6.7.4.2 of the standard. However, when we create a copy of a `restrict`-qualified pointer into a non-`restrict` pointer—as shown in Example 4—this copy becomes *based on* the original and effectively inherits the same properties, even though it is not itself qualified with `restrict`.

This set of considerations led me to make the following proposal.

Proposal

This proposal suggests clarifying the definition of expressions *based on a `restrict` pointer* to include only **rvalue expressions**.

In other words, an expression should be considered "based on a `restrict` pointer" only when it appears as an rvalue — that is, when it is used directly for reading or writing memory without creating new lvalues that might alias.

Valid — direct rvalue access	**Invalid — introducing a new pointer from a restrict expression**

`void foo(int * restrict p) {`
`  *(p + 10) = 10;  // OK`
`}`
	`void foo(int * restrict p) {`
	`  int *x = p + 10;`
	`  *x = 10;  // Undefined behavior`
	`}`

This approach preserves the ability of compilers to perform alias analysis based on `restrict` without requiring complex tracking of all pointer copies and derived expressions in the same scope. It aligns with the original intent of `restrict` — enabling simpler and more predictable optimizations.

Proposed Wording

The wording proposed is a diff from the committee draft of ISO/IEC 9899:2024. **Green text** is new text, while **red text** is deleted text.

Modify 6.7.4.2 paragraph 3 as follows:

> If a declaration of a pointer to an object includes the `restrict` qualifier, and if the block in which the declaration appears allows access to the object only by means of that pointer (and expressions derived from it), then optimization based on the `restrict` qualifier is permitted.
>
> Note that "based on" is defined only for **rvalue expressions with pointer types**.

Modify 6.7.4, examples 5-7:

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 no-aliasing inference based on the parameter declarations alone, though now subtler reasoning is used: that the const-qualification of q precludes it becoming based on p. There is also a requirement that q is not modified, so this alternative cannot be used for the function in Example 2, as written.

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 no-aliasing inference based on the parameter declarations alone, though now even subtler reasoning is used: 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.

EXAMPLE 7 The least effective alternative is:

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

Here the translator can make the no-aliasing 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 described previously. Such a translator is required to assume that aliases are present because 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 consequently need to conservatively assume that aliases are present.

Replace with the following clarified example:

EXAMPLE 5

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

Here, the translator can make the no-aliasing inference because, if `q` becomes based on `p` and there are both a write and a read accesses through `q` and `p`, respectively, the behavior is undefined.

Proposed Wording

The wording proposed is a diff from the committee draft of ISO/IEC 9899:2024. **Green text** is new text, while **red text** is deleted text.

Modify 6.7.4.2 paragraph 3 as follows:

> If a declaration of a pointer to an object includes the `restrict` qualifier, and if the block in which the declaration appears allows access to the object only by means of that pointer (and expressions derived from it), then optimization based on the `restrict` qualifier is permitted.
>
> Note that "based on" is defined only for **rvalue expressions with pointer types**.

Modify 6.7.4, examples 5-7:

> ~~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 no-aliasing inference based on the parameter declarations alone, though now subtler reasoning is used: that the const-qualification of q precludes it becoming based on p. There is also a requirement that q is not modified, so this alternative cannot be used for the function in Example 2, as written.~~
>
> ~~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 no-aliasing inference based on the parameter declarations alone, though now even subtler reasoning is used: 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.

EXAMPLE 7 The least effective alternative is:

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

Here the translator can make the no-aliasing 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 described previously. Such a translator is required to assume that aliases are present because 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 consequently need to conservatively assume that aliases are present.

Replace with the following clarified example:

EXAMPLE 5

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

Here, the translator can make the no-aliasing inference because, if `q` becomes based on `p` and there are both a write and a read accesses through `q` and `p`, respectively, the behavior is undefined.

Acknowledgements

The authors would like to thank the following individuals for their valuable discussions and insights related to this work:

- Konstantin Vladimirov
- Anton Sidorenko
- Sergey Kachkov

References

References

[1] WG14. *Working Draft for the C Standard (N3220)*. Available at: `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf`

[2] Texas Instruments. *Performance Tuning with the* **`restrict`** *Keyword*. Available at: `https://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-files/791/Performance_5F00_Tuning_5F00_with_5F00_the_5F00_RESTRICT_5F00_Keyword.pdf`

[3] WG14. *C99 Defect Report #294: Based-on relationship for restrict-qualified pointers*. Available at: `https://www.open-std.org/jtc1/sc22/wg14/issues/c99/issue0294.html`