ISO/IEC JTC1/SC22/WG21 P3347R1, 2025-01-11

Invalid/Prospective Pointer Operations

Authors: Paul E. McKenney, Maged Michael, Jens Maurer, Peter Sewell, Martin Uecker, Hans Boehm, Hubert Tong, Niall Douglas, Thomas Rodgers, Will Deacon, Michael Wong, David Goldblatt, Kostya Serebryany, Anthony Williams, Tom Scogland, and JF Bastien.
Other contributors: Martin Sebor, Florian Weimer, Davis Herring, Rajan Bhakta, Hal Finkel, Lisa Lippincott, Richard Smith, Chandler Carruth, Evgenii Stepanov, Scott Schurr, Daveed Vandevoorde, Davis Herring, Bronek Kozicki, Jens Gustedt, Peter Sewell, Andrew Tomazos, and Davis Herring.
Audience: SG1, EWG.
Goal: Summarize a proposed solution to enable zap-susceptible concurrent algorithms.

Abstract

The C++ standard currently specifies that all pointers to an object become invalid at the end of its lifetime [basic.life]. This is a software-engineering nightmare because **all** operations on invalid pointers are implementation-defined, even loads and stores. This means that concurrent algorithms such as LIFO Push that knowingly use invalid pointers must have (for example) converted such pointers to `uintptr_t` *before* they became invalid. This requirement can cause `uintptr_t` to spread throughout unrelated portions of a program using algorithms such as LIFO Push, which is but one example of the aforementioned software-engineering nightmare.

We therefore propose that all pointer arithmetic operations faithfully compute value representation, even those involving invalid pointers and pointers having prospective provenance. Note that comparisons and dereference operations on invalid pointers remain implementation-defined.

Background

Section 7.3.2 (“Lvalue-to-rvalue conversion” [conv.lval]) p3.3 reads as follows:

> Otherwise, if the object to which the glvalue refers contains an invalid pointer value (6.7.5.5.3), the behavior is implementation-defined.

This means that an implementation is permitted to (for example) return a random number from a load of an invalid pointer.

Although implementation-defined loads from, stores to, and arithmetic on invalid pointers might permit additional diagnostics and optimizations, it is not consistent with long-standing usage, especially for a range of concurrent and sequential algorithms that rely on loads, stores, equality comparisons, and even dereferencing of such pointers. Similar issues result from object-lifetime aspects of C++ *pointer provenance*. These problems will be largely (but not completely) solved if P2434R2: Nondeterministic pointer provenance and P2414R4 Pointer lifetime-end zap proposed solutions are adopted into the IS. This paper assumes that these papers are adopted, and builds on the angelic non-determinism and additional ergonomics described in those papers by defining loads, stores, and arithmetic operations on invalid and provisional pointers, as is required in order for certain concurrent algorithms to gain benefit from angelic provenance. However, comparisons and dereferences retain their current implementation-defined and undefined status.

Please see the papers listed in the History section for more background information on pointer zap and its relation to angelic provenance.

What We Are Asking For

We propose that all non-comparison non-dereference computations involving pointers, including normal loads and stores, must faithfully compute the value representation, even if the pointers are invalid or contain

prospective values. If the initial pointers contain prospective values, the resulting pointers will also contain prospective values.

In particular, the standard must require that implementations are not allowed to modify the pointer's value representation in response to the end of the lifetime of the pointed-to object. This tightens the implementation-defined behavior of applying various operations to invalid pointers that are needed in order to enable portable code, for example, as discussed in the "Consequences for pointer zap" section of P2434R1.

Possible poll:

1. Do we want to tighten the implementation-defined behavior of applying non-comparison non-dereference operation to invalid and provisional pointers so that value representations are generated as if the pointers were valid, for example, as discussed in the "Consequences for pointer zap" section of P2434R1?
2. Do we want to tighten the implementation-defined behavior of applying non-comparison non-dereference operation to invalid and provisional pointers so that if the initial pointers contained prospective values, then the resulting pointers will also contain prospective values?

Detailed Proposal

This section describes the tightening of the implementation-defined behavior of applying various operations to invalid and provisional pointers that is needed in order to enable portable code, for example, as discussed in the "Consequences for pointer zap" section of P2434R1.

Non-comparison non-dereference computations involving pointers, including normal loads and stores, must faithfully compute the value representation, even if the pointers are invalid or provisional. In particular, implementations are not allowed to modify the pointer's value representation in response to the end of the lifetime of the pointed-to object.

First, the implementation must actually execute the corresponding load or store instruction, give or take optimizations that fuse or invent load and stores or that eliminate dead code. Note that load and store operations include passing of parameters and returning of values.

Second, non-comparison arithmetic operations must produce value representation consistent with those of their operands. For example, adding an integral constant to an invalid pointer must result in the same value representation as would that same addition to a valid pointer of the same type having the same initial value representation.

Third, comparison operations must be deterministic, that is, successive comparisons of a pair of pointers A and B must give consistent results. Note that this applies only so long as each of the pointers A and B remain identical. In particular, suppose that a pointer C is derived from pointer A, but with provenance recomputed, perhaps due to having traversed a translation-unit boundary. Then there is no requirement that comparisons of A and B be consistent with comparisons of C and B.

Finally, note that any remaining implementations that use trap representations for pointers need special attention, at least assuming that there is any such hardware that is using modern C++ implementations. Alternatives include:

1. Remove support for platforms having trappable pointer values. This approach is best if there are no longer any such platforms, or if all such platforms will continue to use old compiler versions.
2. Support trappable pointer values using some non-standard extension, for example, using a command-line argument for that purpose (or an environment variable, or a compiler-installation option, or a special build of the compiler, or similar). Note that programs relying on trappable pointer values are already non-portable, so this approach does not place additional limits on such programs.
3. Modify the standard to provide explicit syntax for trappable pointers. This approach would require changes to existing programs that rely on trappable pointers, but such changes might provide great documentation benefits and might also be quite useful to tools carrying out pointer-based formal verification.

Please note that trap representations for pointers are not mainstream. Current hardware such as ARM MTE instead reserve pointer bits that can be thought of as approximating provenance information. Please also note that (as of March 23, 2024), the proposed resolution to CWG2822 Side-effect-free pointer zap makes it clear that the end of an object's lifetime does not affect the value representation of pointers to that object, which is a welcome step in the right direction.

Wording

Referencing N4993 C++ Working Draft:

- Section 7.3.2 ("Lvalue-to-rvalue conversion" [conv.lval]) p3.3:
  Otherwise, if the object to which the glvalue refers contains an invalid pointer value (6.7.5.5.3), the behavior is implementation-defined but the value representation of the rvalue shall be consistent with those of the glvalue.
- Section 6.7.3 ("Lifetime" [basic.life]) p6 should not need change:
  Before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that represents the address of the storage location where the object will be or was located may be used but only in limited ways. …
- Section 6.7.3 ("Lifetime" [basic.life]) p7 similarly need not change:
  Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways.
- Section 6.7.3 ("Lifetime" [basic.life]) p9 similarly need not change:
  After the lifetime of an object has ended and before the storage which the object occupied is reused or released, if a new object is created at the storage location which the original object occupied and the original object was transparently replaceable by the new object, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object.
- Section 6.8.4 ("Compound types" [basic.compound]) p4 is covered by P2434R2 Nondeterministic pointer provenance.

Searches: "invalid pointer value", "invalid pointer", "safely-derived pointer" (none), "become invalid" (none),
Immutable searches: [basic.stc], [basic.life],

History

P3347R1:

- In response to November 22, 2024 Wroclaw EWGI review:
  - Sharpen abstract, moving details into a new “Background” section.
  - Updated from “representation bytes” to “value representation” to track N4993 C++ Working Draft.
- Update paper references.

P3347R0:

- Extracted from D2414R4 to allow this content to move separately to EWG.

D2414R4:

- Updated based on the June 24, 2024 St. Louis EWG review and forwarding of “P2434R1: Nondeterministic pointer provenance” from Davis Herring and subsequent discussions:
  - The prospective-pointer semantics remove the need for a provenance fence, but add the need for a definition of “prospective pointer”.
  - Leverage prospective pointer values.
  - Adjust example code accordingly.

P2414R3:

- Includes feedback from the March 20, 2024 Tokyo SG1 and EWG meetings, and also from post-meeting email reflector discussions.
- Change from reachability to fence semantic, resulting in provenance_fence().
- Add reference to C++ Working Draft [basic.life].

P2414R2:

- Includes feedback from the September 1, 2021 EWG meeting.
- Includes feedback from the November 2022 Kona meeting and subsequent electronic discussions, especially those with Davis Herring on pointer provenance.
- Includes updates based on inspection of LIFO Push algorithms in the wild, particularly the fact that a LIFO Push library might not have direct access to the stack node’s pointer to the next node.
- Drops the options not selected to focus on a specific solution, so that P2414R1 serves as an informational reference for roads not taken.
- Focuses solely on approaches that allow the implementation to reconsider pointer invalidity only at specific well-marked points in the source code.

P2414R1 captures email-reflector discussions:

- Adds a summary of the requested changes to the abstract.
- Adds a forward reference to detailed expositions for atomics and volatiles to the “What We Are Asking For” section.
- Add a function `atomic_usable_ref` and change `usable_ptr::ref` to `usable_ref`. Change A2, A3, and Appendix A accordingly.
- Rewrite of section B5 for clarity.

P2414R0 extracts and builds upon the solutions sections from P1726R5 and P2188R1. Please see P1726R5 for discussion of the relevant portions of the standard, rationales for current pointer-zap semantics, expositions of prominent susceptible algorithms, the relationship between pointer zap and both happens-before and value-representation access, and historical discussions of options to handle pointer zap.

The WG14 C-Language counterparts to this paper, N2369 and N2443, have been presented at the 2019 London and Ithaca meetings, respectively.

Appendix: Relationship to WG14 N2676

WG14's N2676 "A Provenance-aware Memory Object Model for C" is a draft technical specification that aims to clarify pointer provenance, which is related to lifetime-end pointer zap. This technical specification puts forward a number of potential models of pointer provenance, most notably PNVI-ae-udi. This model allows pointer provenance to be restored to pointers whose provenance has previously been stripped (for example, due to the pointer being passed out of the current translation unit as a function parameter and then being passed back in as a return value), but the restored provenance must correspond to a pointer that has been *exposed*, for example, via a conversion to integer, an output operation, or direct access to that pointer's value representation.

Note that `compare_exchange` operations access a pointer's value representation, and thus expose that pointer. We recommend that other atomic operations also expose pointers passed to them. We also note that given modern I/O devices that operate on virtual-address pointers (using I/O MMUs), volatile stores of pointers must necessarily be considered to be I/O, and thus must expose the pointers that were stored. In addition, either placing a pointer in an object of type `usable_ptr<T>` or accessing a pointer as an object of type `usable_ptr<T>` exposes that pointer. Finally, note that the changes recommended by N2676 would make casting of pointers through integers a good basis for the `usable_ptr<T>` class template.

We therefore see N2676 as complementary to and compatible with pointer lifetime-end zap. We do not see either as depending on the other.}

Appendix: Relation to WG21 P2434R2

WG21's "P2434R2: Nondeterministic pointer provenance" proposes refinements to the definition of pointer zap. This current paper does not conflict with that paper, but rather builds on top of that paper in order to provide more ergonomics for users.