ISO/IEC JTC1/SC22/WG21 P3347R3, 2025-07-11

Invalid Pointer Operations

Authors: Paul E. McKenney (paulmckrcu@kernel.org), 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, JF Bastien, and Jason McGuiness.
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, and Andrew Tomazos.
Audience: CWG.
Goal: Summarize a proposed pointer load/store/assign/copy 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 non-arithmetic pointer operations faithfully compute value representation, even those involving invalid pointers. 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 and stores to 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 P2414R4 Pointer lifetime-end zap proposed solutions is adopted into the IS. This paper assumes that this paper is adopted, and builds on the additional ergonomics described in that paper by defining loads and stores on invalid and prospective pointers, as is required in order for certain concurrent algorithms. However, comparisons and dereferences retain their current implementation-defined and undefined status, respectively.

Please see the papers listed in the History section for more background information on pointer zap.

What We Are Asking For

We propose that all non-comparison non-arithmetic non-dereference computations involving pointers, specifically including normal loads and stores, are fully defined even if the pointers are invalid.

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-arithmetic non-dereference operations (specifically including load and store) to invalid and prospective pointers so that these operations are defined even if the pointers are invalid, for example, as discussed in the “Consequences for pointer zap” section of P2434R4?

Detailed Proposal

This section describes the tightening of the implementation-defined behavior of applying various operations to invalid and prospective 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-arithmetic non-dereference computations involving pointers, specifically including normal loads and stores, are defined even if the pointers are invalid or prospective. Note that since the adoption of CWG2822, implementations are not allowed to modify the pointer’s value representation in response to the end of the lifetime of the pointed-to object.

In other words, 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. These operations are defined, even on invalid or prospective pointers. We believe that this affects only initialization, comparison, and the various conversions (lvalue-to-rvalue, const_cast, reinterpret_cast, etc.).

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. Construct the implementation such that non-pointer facilities are used to load pointer values when the implementation cannot prove that the value will be dereferenced. We are told that trappable-pointer implementations already use this technique.
2. Construct the implementation such that the trap handler does any needed fixups and resumes execution in cases where the trap is due to a pointer-value load when the implementation cannot prove that this value will be dereferenced. We do not know of any such implementation, and we do not expect to ever hear of one. We nevertheless include this approach for completeness.
3. 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.
4. 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.

5. 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 approximate provenance information, and thus handles this proposal as-is. 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.

Additional proposals for other aspects of the pointer-zap problem are in P2414R9 ("Pointer lifetime-end zap proposed solutions: atomics and volatile") and PNNNNR0 ("Pointer lifetime-end zap proposed solutions: bag-of-bits pointer class").

Example: LIFO Push With Direct Pointer Access

One form of LIFO push implementation provides direct access to the `Node` class's `.next` pointer:

```
template <typename Node> class LIFOList { // Node must support set_next()
  std::atomic<Node*> top_{nullptr};
 public:
  void push(Node* newnode) {
    newnode->next = top_.load(); // step 1
    while (true) {
      if (top_.compare_exchange_weak(newnode->next, newnode->next)) return; // step 2
    }
  }

  Node* pop_all() { return top_.exchange(nullptr); }
};
```

Given the current standard, the above code is buggy because it is subject to lifetime-end pointer zap.

Use Case 1: Invalid Pointer Use

The following sequence of events illustrates an invalid-pointer vulnerability given the current C++ standard:

- `top_` holds pointer to node `X1` at location `A`.
  `top_` --> `A` (address of `X1`)
- Thread T1 executes step 1 of `push(&X2)`.
  `X2.next_` --> `A` (address of `X1`)
- Thread T2 executes `pop_all`, deletes `X1`.
  `X1` deleted
  `top_` --> `null`
  `X2.next_` --> `A` (address of `X1`) <<<<< **invalid pointer!!!**

- Thread T1 executes step 2 of `push(&X2)` and **uses invalid pointer** `X2.next` in `.compare_exchange_weak`.

Use Case 2: Zombie Pointer Dereference

The following sequence of events illustrates a zombie-pointer vulnerability given the current C++ standard:

- `top_` holds pointer to node `X1` at location `A`.
  `top_` --> `A` (address of `X1`)
- Thread T1 executes step 1of `push(&X2)`.
  `X2.next_` --> `A` (address of `X1`)
- Thread T2 executes `pop_all`, deletes `X1`.
  `X1` deleted
  `top_` --> `null`
  `X2.next_` --> `A` (address of `X1`) <<<<< **invalid pointer!!!**
- Thread T2 allocates node `X3` that happens to be at location `A`, and executes `push(&X3)`
  `top_` --> `A` (address of `X3`)
  `X2.next_` --> `A` (address of `X1` and `X3`) <<<<< **zombie pointer!!!**
- Thread T1 executes step 2 of `push(&X2)` and **uses invalid pointer** `X2.next` in `.compare_exchange_weak`, but let's ignore that complication and note that the bitwise comparison will cause this to succeed:
  `top_` --> `&X2`
  `X2.next_` --> `A` (address of `X1` and `X3`) <<<<< **zombie pointer!!!**
- Thread T1 executes `pop_all`, dereferences `X2.next_`, which holds value `A` (address of `X1` and `X3`), i.e., a zombie pointer.

Fixing LIFO Push Using This Proposal

Assuming the changes put forward by this proposal and those in D2414R8 ("Pointer lifetime-end zap proposed solutions: atomics and volatile"), the required source-code changes are highlighted in yellow:

```
template <typename Node> class LIFOList { // Node must support set_next()
  std::atomic<Node*> top_{nullptr};
 public:
  void push(Node* newnode) {
    newnode->next = top_.load(); // step 1
    while (true) {
      if (top_.compare_exchange_weak(newnode->next, newnode->next)) return; // step 2
    }
  }

  Node* pop_all() { return top_.exchange(nullptr); }
};
```

Note that there is no code highlighted in yellow. This is fixed because the load and assignment operations are now defined and because the `.compare_exchange_weak` operation conceptually rewrites the newnode->next pointer even in the success case, providing current provenance.

- `top_` holds pointer to node `X1` at location `A`.
  `top_` --> `A` (address of `X1`)
- Thread T1 executes step 1 of `push(&X2)`.
  `X2.next_` --> `A` (address of `X1`)
- Thread T2 executes `pop_all`, deletes `X1`.
  `X1` deleted
  `top_` --> `null`
  `X2.next_` --> `A` (address of `X1`) <<<<< **invalid pointer!!!**
- Thread T1 executes step 2 of `push(&X2)` and **uses invalid pointer** `X2.next` in `.compare_exchange_weak`. Which is OK because the various conversion and initialization operations are now defined and because the `.compare_exchange_weak` operation deals in bits, and not in any provenance that is unrepresented in those bits.

The second use case is fixed because the `.compare_exchange_weak` operation is now defined to conceptually overwrite its second argument even when this operation succeeds. Please note that this strictly conceptual overwrite affects only provenance information, and does not cause any actual code to be emitted:

- `top_` holds pointer to node `X1` at location `A`.
  `top_` --> `A` (address of `X1`)
- Thread T1 executes step 1of `push(&X2)`.
  `X2.next_` --> `A` (address of `X1`)
- Thread T2 executes `pop_all`, deletes `X1`.
  `X1` deleted
  `top_` --> `null`
  `X2.next_` --> `A` (address of `X1`) <<<<< **invalid pointer!!!**
- Thread T2 allocates node `X3` that happens to be at location `A`, and executes `push(&X3)`
  `top_` --> `A` (address of `X3`)
  `X2.next_` --> `A` (address of `X1` and `X3`) <<<<< **zombie pointer!!!**
- Thread T1 executes step 2 of `push(&X2)` and **uses invalid pointer** `X2.next` in `.compare_exchange_weak`, but as noted above, this is now OK. Also note that the bitwise comparison will cause this to succeed:
  `top_` --> `&X2`
  `X2.next_` --> `A` (address `X3`) <<<<< **no longer zombie pointer!!!**
- Thread T1 executes `pop_all`, dereferences `X2.next_`, which holds value `A` (now just the address `X3`), i.e., a perfectly usable pointer.

Wording

Referencing [N4993 C++ Working Draft]:

- Section 6.8.4 (“Compound types” [basic.compound]) p4:
  A pointer value P is valid in the context of an evaluation E if P is a null pointer value, or if it is a pointer to or past the end of an object O and E happens before the end of the duration of the region of storage for O. If a pointer value P is used in an evaluation E and P is not valid in the context of E~~, then the behavior is undefined if E is an~~

~~indirection (7.6.2.2) or an invocation of a deallocation function (6.7.5.5.3), and implementation-defined otherwise.~~:

- If, in E, P is the operand of the indirection operator ([expr.unary.op]), the behavior is undefined.
- If, in E, P is either
  - subjected to a boolean conversion ([conv.bool]), or
  - is the operand of a unary +, additive, three way comparison, relational, or equality operator ([expr.compound]),

  the behavior is implementation-defined.

- Section 7.3.2 (“Lvalue-to-rvalue conversion” [conv.lval]) delete 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 result still contains an invalid pointer value.~~

History

D3347R3:

- Pull in non-direct-pointer access example.
- Polls in EWG resulted in unanimous consent to forward to CWG.
- Core wording feedback (AKA total rewrite of wording) from Jens Maurer and Davis Herring.
- Add alternatives for pointer trap representations based on discussions with other old-timers.

P3347R2:

- In response to discussions at the 2025 Hagenberg meeting:
    - Drop the pointer-arithmetic requirement. Note that this requirement cannot be met for incomplete types, whose size is unknown to the compiler. This requirement might be added later, but restricted to non-incomplete types.
    - Remove all references to angelic provenance.
    - Remove all references to prospective pointers.
        - References to “angelic” and “prospective” might be added back if a later version of “P2434R2 Nondeterministic pointer provenance” moves forward.

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.