ISO/IEC JTC1/SC22/WG21 P2414R2, 2023-12-17

Pointer lifetime-end zap proposed solutions

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, and Anthony Williams.
Other contributors: Martin Sebor, Florian Weimer, Davis Herring, Rajan Bhakta, Hal Finkel, Lisa Lippincott, Richard Smith, JF Bastien, Chandler Carruth, Evgenii Stepanov, Scott Schurr, Daveed Vandevoorde, Davis Herring, Bronek Kozicki, Jens Gustedt, Peter Sewell, and Andrew Tomazos.
Audience: SG1, EWG.
Goal: Summarize 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. Although this permits additional diagnostics and optimizations which might be of some value, 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*.

We propose (1) all non-comparison non-dereference computations involving pointers, including normal loads and stores, must faithfully handle the representation bytes, even if the pointers are invalid; (2) the addition to the C++ standard library of the class template std::usable_ptr<T> that is a pointer-like type that is still usable after its pointee's lifetime has ended (with usability extended to pointers reachable from that pointer); and (3) that atomic and volatile operations be redefined to forgive lifetime-end pointer invalidity.

Please note that this paper does not propose adding bag-of-bits pointer semantics to the standard. However, in the service of legacy code, it is hoped that implementers provide some facility such as a command-line option that causes all pointers to be exempt from lifetime-end pointer invalidity.

History

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 representation-byte 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.

Introduction

The C language has been used to implement low-level concurrent algorithms since at least the early 1980s, and C++ has been put to this use since its inception. However, low-level concurrency capabilities did not officially enter either language until 2011. Given about 30 years of independent evolution of C and C++ on the one hand and concurrency on the other, it should be no surprise that some corner cases were missed in the efforts to add concurrency to C11 and C++11.

A number of long-standing and heavily used concurrent algorithms, one of which is presented in a later section, involve loading, storing, casting, and comparing pointers to objects which might have reached their lifetime end between the pointer being loaded and when it is stored, reloaded, cast, and compared, due to concurrent removal and freeing of the pointed-to object. In fact, some long-standing algorithms even rely on dereferencing such pointers, but in C++, only in cases where another object of similar type has since been allocated at the same address. This is problematic given that the current standards and working drafts for both C and C++ do not permit reliable loading, storing, casting, or comparison of such pointers. To quote Section 6.2.4p2 (“Storage durations of objects”) of the ISO C standard:

> The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime. (See WG14 N2369 and N2443 for more details on the C language’s handling of pointers to lifetime-ended objects.)

However, (1) concurrent algorithms that rely on loading, storing, casting, and comparing such pointer values have been used in production in large bodies of code for decades, (2) automatic recognition of these sorts of algorithms is still very much a research topic (even for small bodies of code), and (3) failures due to non-support of the loading, storing, comparison, and (in certain special cases) dereferencing of such pointers can lead to catastrophic and hard-to-debug failures in systems on which we all depend. We therefore need a solution that not only preserves valuable optimizations and debugging tools, but that also works for existing source code. After all, any solution relying on changes to existing software systems would require that we have a way of locating the vulnerable algorithms, and we currently have no such thing.

This is not a new issue: the above semantics have been in the C standard since 1989, and the algorithm called out below was put forward in 1973. But its practical consequences will become more severe as compilers do more optimisation, especially link-time optimisation, and especially given the ubiquity of multi-core hardware.

This paper proposes specific solutions. Currently, these either incur runtime overhead on the one hand or use type punning on the other. Although type punning should be permissible in carefully controlled library code, potential solutions involving neither type punning nor runtime overhead would be welcome.

Terminology

- *Bag of bits:* A simple model of a pointer consisting only of its associated address and type, excluding any additional information that might be gleaned from lifetime-end pointer zap and pointer provenance. A simple compiler might well model its pointers as bags of bits. For the purposes of this paper, a non-simple compiler can be induced to treat pointers as bags of bits by marking all pointer accesses and indirections as `volatile`, albeit with possible performance degradation.

- *Invalid pointer:* A pointer referencing an object whose storage duration has ended. For more detail, please see the "What Does the C++ Standard Say?" section of P1726R5, particularly the reference to section 6.7.5p4 [basic.stc] of the standard ("When the end of the duration of a region of storage is reached, the values of all pointers representing the address of any part of that region of storage become invalid pointer values"). In the C standard, such a pointer is termed an *indeterminate pointer*.
- *Invalid pointer use*: Any use of an invalid pointer (including reading, writing, comparison, casting, and passing to a non-deallocation function) other than passing it to a deallocation function and indirection through it. [Intended to correspond to [basic.stc]p4 "*Any other use of an invalid pointer value has implementation-defined behavior.*"]
- *Lifetime-end pointer zap:* An event causing a pointer to become invalid, or, in WG14 parlance, indeterminate. Because this is a WG21 document, the term *becomes invalid* is used in preference to "lifetime-end pointer zap", however, text that needs to cover both C++ and C will use the term "lifetime-end pointer zap", "pointer zap", or just "zap".
- *Pointer provenance:* Implementations are permitted to model pointers as more than just a bag of bits.
- *Simple compiler:* A compiler that does no optimization. For the purposes of this paper, results similar to those of a simple compiler can be obtained by treating all pointers as bags of bits.
- *Zap-susceptible algorithm:* An algorithm that relies on invalid pointer use and/or zombie pointer dereference.
- *Zombie pointer:* An invalid pointer that happens to contain the same address as a currently valid pointer to an object of compatible type.
- *Zombie pointer dereference*: Indirection through a zombie pointer. [The relevant part of the standard being [basic.stc] p4: "*Indirection through an invalid pointer value and passing an invalid pointer value to a deallocation function have undefined behavior.*"]

What We Are Asking For

In order to support a number of critically important algorithms, this paper proposes a `usable_ptr<T>` template class to mark pointers in order to forgive pointer invalidity both of that pointer and also of any pointers reachable from that pointer (details in sections A2 and B2).

Note that this paper does not propose blanket bag-of-bits pointer semantics, despite a great many users being strongly in favor of such semantics (P2188R1). It is therefore hoped that implementers will provide some facility to cause pointers to be treated as bags of bits from a pointer-invalidity viewpoint, perhaps by implicitly treating all pointer types as if they were `usable_ptr<T>`. This would be helpful for legacy code.

In addition, this paper proposes that non-comparison non-dereference computations involving pointers, including normal loads and stores, must faithfully handle the representation bytes, even if the pointers are invalid. In particular, the standard must require that implementations are not allowed to modify the pointer's representation bytes in response to the end of the lifetime of the pointed-to object.

This paper additionally proposes that atomic operations have the side effect of forgiving pointer invalidity (details in next section).

Furthermore, this paper also notes that `volatile` accesses must necessarily forgive invalidity in order to support passing of virtual addresses to and from I/O devices, which has long been supported in hardware, either by virtue of that hardware lacking any sort of memory-management unit (MMU) or that hardware being equipped with an I/O MMU that maps addresses provided by hardware devices.

Finally, this paper notes that the implementation must prove that a given pointer is invalid before taking action based on invalidity.

The following sections provide more detail on this proposal and also of the options considered since P1726R4. Those interested in seeing a wider array of historical options are invited to review P1726R5 and P2188R1.

Detailed Proposal

As noted earlier, this paper proposes: (1) Faithful computation of representation bytes of invalid pointers, (2) A `usable_ptr<T>` template class, (3) That atomic operations have the side effect of forgiving pointer invalidity, and (4) that `volatile` accesses continue to forgive pointer invalidity.

Faithful Computation of Representation Bytes of Invalid Pointers

Non-comparison non-dereference computations involving pointers, including normal loads and stores, must faithfully handle the representation bytes, even if the pointers are invalid. In particular, the standard must require that implementations are not allowed to modify the pointer's representation bytes 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. Note that load and store operations include parameter passing and return values.

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

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, not that any remaining implementations that use trap representations for pointers need special attention. 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 …). 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 trapping pointers are not mainstream. Current proposals such as Cheri instead reserve pointer bits that can be thought of as carrying approximations of provenance information.

A `usable_ptr<T>` Template Class

A `usable_ptr<T>` template class may be used to mark pointers in order to forgive pointer invalidity. The provenance discussion gives a solid basis for this, but there is a need to treat normal user-supplied pointers as if they were of the `usable_ptr<T>` template class.

In addition a `recursive_recompute_provenance()` function recomputes provenance for the pointer passed in (returning a copy of that pointer with recomputed provenance), but also recomputes provenance of any pointer reached from that pointer. This does not restrict current compiler optimizations because the compiler either:

1. Knows about the call to `usable_ptr<T>` or
2. Knows nothing about the pointer, and thus cannot apply any preconceived provenance. Please keep in mind that the compiler is not permitted to invent pointer comparisons. However, if the user compares pointers, the compiler is of course permitted to draw reasonable conclusions regarding provenance, especially if the pointers compare equal.

The recursive nature proposed for the `recursive_recompute_provenance()` function addresses existing use cases where the LIFO Push library does not have direct access to the pointer from one stack node to the next. Note that in both GCC and Clang/LLVM, the following statement has the desired effect:

```
__asm__ __volatile__("": : :"memory");
```

This might restrict compiler optimizations more than is necessary, so a lighter-weight mechanism would be welcome. However, this memory clobber has the virtue of already existing and having been used heavily for some decades.

The name `usable_ptr<T>` has been criticized as not being particularly illuminating. Perhaps something like `reevaluate_provenance<T>`, `regenerate_provenance<T>`, `recompute_provenance<T>`, `update_provenance<T>`, or similar would be better. But what is in a name?

Atomic Operations Forgive Pointer Invalidity

Atomic operations have the side effect of forgiving pointer invalidity. One way to think of this (due to Davis Herring) is that values stored in atomic pointers are treated as if the member of the `atomic<T*>` type holding the pointer value is of integral type, with each access to that pointer value involving an appropriate cast. This means that provenance is re-evaluated whenever a pointer is loaded from an `atomic<T*>` object. It also means that whenever a pointer is stored to an `atomic<T*>` object, the implementation treats that pointer as having been exposed.

Although concerns were raised at the 2022 Kona meeting about possible optimization limitations from this approach. However, given a common-case atomic object, any thread might possibly store into it at any time, making tracking of provenance through atomic pointers of dubious utility at best.

Previous discussions have put forward the notion of “flattening” optimizations that combine all threads into a single thread, with the notion that the implementation might perform exact analysis of this single thread. However, such optimizations can generate infinite loops and deadlocks that would not be present in the original multithreaded code. Given the oracular analysis required to make flattening work for locking and polled atomic operations, the additional analysis required to forgive invalidity for atomic pointers should not be at all difficult by comparison.

Whenever a reference to a pointer value is used as the old value by a CAS operation (even a successful one that might not be considered to modify the old value), any prior provenance associated with that pointer value is discarded.

As soon as a value is loaded from an atomic pointer, the resulting non-atomic pointer is immediately subject to any future lifetime-end pointer invalidity. However, as noted earlier, implementations are not permitted to allow this invalidity to affect the values of the representation bytes.

Volatile Accesses Forgive Pointer Invalidity

Note that `volatile` accesses must necessarily forgive invalidity in order to support passing of virtual addresses to and from I/O devices. To see this, keep firmly in mind that the OS kernel (written in C or C++) is communicating via memory with device firmware (also written in C or C++).

Therefore, a `volatile` load from a pointer should be treated as if that pointer was of integral type with appropriate cast to the pointer’s type. This causes provenance to be re-evaluated, forgiving any prior invalidity. This forgiveness is at that point in time only, ending as soon as the pointer value is placed in a non-volatile/non-atomic object. Similarly, a `volatile` store to a pointer should be treated as if that pointer was of integral type with appropriate cast to the integral type. In particular, any invalidity of the pointer stored is forgiven.

However, just as for atomic operations, as soon as any value is obtained from a `volatile` load, the resulting non-`volatile` pointer is immediately subject to any future lifetime-end pointer invalidity. Also just as for atomic operations, implementations are not permitted to allow this invalidity to affect the values of the representation bytes.

Examples

LIFO Push

A simple atomic LIFO Push algorithm is as follows:

```
template <typename Node> class LIFOList { // Node must support set_next()
  std::atomic<Node*> top_{nullptr};
 public:
  void push(Node* newnode) {
    while (true) {
      Node* oldtop = top_.load(); // step 1
      newnode->set_next(oldtop); // step 2
      if (top_.compare_exchange_weak(oldtop, newnode) return; // step 3
    }
  }

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

Again, note the use of the `set_next()` member function as opposed to direct access to the pointer linking the nodes in the stack. This idiom is used in the wild, for example, in cases where instrumenting this member function assists with debugging and performance-analysis tasks.

This code 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 steps 1 and 2 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`)
- Thread T1 executes step 3 of `push(&X2)` and **uses invalid pointer** `A` 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 steps 1 and 2 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`)
- 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 3 of `push(&X2)` and `.compare_exchange_weak` succeeds

    top_ --> &X2
    X2.next_ --> A (address of X1 and X3)

- 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

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) {
    while (true) {
      Node* oldtop = top_.load(); // step 1
      newnode->set_next(oldtop); // step 2
      if (top_.compare_exchange_weak(oldtop, newnode) return; // step 3
    }
  }

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

The first use case is fixed due to the requirement that pointer invalidity not modify representation bytes:

- top_ holds pointer to node X1 at location A.
  top_ --> A (address of X1)
- Thread T1 executes steps 1 and 2 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)
- Thread T1 executes step 3 of push(&X2) and uses invalid pointer A in .compare_exchange_weak. However, this atomic operation looks only at representation bytes, which must not be unaffected by pointer invalidity, which fixes this example.

The second use case is fixed by the use of usable_ptr<T>:

- top_ holds pointer to node X1 at location A.
  top_ --> A (address of X1)
- Thread T1 executes steps 1 and 2 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)

- 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) <<<<< still a zombie pointer
- Thread T1 executes step 3 of push(&X2) and .compare_exchange_weak succeeds
  top_ --> &X2
  X2.next_ --> A (address of X1 and X3)
- Thread T1 executes pop_all, dereferences X2.next_, which holds value A (address of X1 and X3), i.e., a zombie pointer. However, the usable_ptr<T> forces the current provenance for that address (X3) to be used, thus repairing this use case.

These two examples demonstrate use of the changes proposed in this paper.

User Tracking of Pointers and realloc()

Hans's realloc() example compares the return value of realloc() with its argument to determine whether other pointers to the pointed-to object need to be updated. Here is Hans's original code:

```
q = realloc(p, newsize);
if (q != p)
      update_my_pointers(p, q);
```

This code can be simplified as follows:

```
T* p;

q = realloc(p, newsize);
if (q != p)
      p = q;
```

And then this simplified code can be fixed using usable_ptr<T> as follows:

```
usable_ptr<T> p;

q = realloc(p, newsize);
if (q != p)
   p = q;
```

Using this proposal, one could add a line "p = usable_ptr<p>" immediately after the realloc() call. In addition, each of the copies of p updated by update_auxiliary_pointers() must be subjected to usable_ptr(). This requires that re-evaluating provenance on invalid pointers that reference freed memory be compared according to their representation bytes, an additional requirement not called out above.

Appendix: Prototype `usable_ptr<T>` Implementation

```
#include <atomic>
#include <iostream>

void recursive_recompute_provenance() { __asm__ __volatile__("": : :"memory"); }
```

The memory-clobber asm is overkill. If no lighter-weight mechanism (such as a no-op) is available, then there might need to be two different flavors of the get() member function, one with the recursive provenance recomputation and one without.

This implementation assumes that any C++ implementations that might currently track pointer provenance through integer conversions stop doing so. Note that `usable_ptr<T>` may be trivially implemented in terms of `intptr_t` or `uintptr_t` on the one hand or `Atomic<T*>`or the other, but with only constructors, destructors, assignment, and indirection member functions exported.

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 representation.

Note that `compare_exchange` operations access a pointer's 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 P2434R0

WG21's "P2434R0: Nondeterministic pointer provenance" proposes refinements to the definition of pointer zap. This current paper does not conflict with that paper, but rather provides ways for the user to avoid pointer zap.