ISO/IEC JTC1/SC22/WG21 P2414R4, 2024-08-12

# 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, Anthony Williams, Tom Scogland, JF Bastien, and Daniel Krügler.
**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, 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]. 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) the addition to the C++ standard library of the function make_ptr_prospective() that takes a pointer argument and returns a prospective pointer value corresponding to its argument; (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 the pointed-to object's lifetime has ended; (3) that atomic operations be redefined to yield and to store prospective pointers values; and (4) that operations on volatile pointers be defined to yield and to store prospective pointer values.**

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 such semantics, perhaps via some facility such as a command-line option that causes all pointers to be exempt from lifetime-end pointer invalidity.

## 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 decades 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 and WG21 P1726R5 for the corresponding C++ language details.)

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 this issue's 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 straightforward specific solutions.

## 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 6.7.5.1p4 [basic.stc.general] 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, passing to a non-deallocation function), and indirection through it. [Intended to correspond to [basic.stc.general] 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.
- *Prospective pointer value:*  A pointer value corresponding to an object whose lifetime has not started, including a pointer to an object whose region of storage has not yet been created.  A correct algorithm will not compare or dereference a prospective pointer until after an appropriately typed object's lifetime starts at the address indicated by the pointer's value.  Note that comparison of a prospective pointer's representation bytes is permitted, for example, as carried out by the **`.compare_exchange`** member function.  One way to produce a prospective pointer is to cast a valid pointer to `uintptr_t` and then cast it back to the same pointer type.  Support of prospective pointers is optional for implementations that do not provide `uintptr_t`.  For more information, please see P2434R1.
- *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 whose representation bytes happen to correspond to the same memory 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.general] 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 `make_ptr_prospective()` function and a `usable_ptr<T>` template class to provide a convenience mechanism for encapsulating the pair of `reinterpret_cast<>` operations used to create a prospective pointer value as described in the “Consequences for pointer zap” section of P2434R1.

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.

This paper additionally proposes a convenience mechanism in which `std::atomic<T*>` and `volatile` have special behavior so that the associated pointer values become prospective, in the former case, as alluded to in the “Consequences for pointer zap” section of P2434R1.

Furthermore, this paper also proposes that `volatile` accesses 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. For example, consider a device whose firmware and driver are both written in C++.

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.

Possible polls:

1. Do we want a `make_ptr_prospective()` function that provides a convenience mechanism for encapsulating the pair of reinterpret_cast<> operations used to create a prospective pointer value as described in the “Consequences for pointer zap” section of P2434R1?
2. Do we want a `usable_ptr<T>` convenience mechanism for encapsulating the pair of reinterpret_cast<> operations used to create a prospective pointer value as described in the “Consequences for pointer zap” section of P2434R1?
3. Do we want a convenience mechanism in which `std::atomic<T*>` has special behavior so that the associated pointer values become prospective, as alluded to in the “Consequences for pointer zap” section of P2434R1?
4. Do we want a convenience mechanism in which `volatile` has special behavior so that the associated pointer values become prospective?

## Detailed Proposal

As noted earlier, this paper proposes: (1), `make_ptr_prospective()` function (2) A `usable_ptr<T>` template class, and (3) That atomic operations have the side effect of forgiving pointer invalidity, and (4) that `volatile` accesses have the side effect of forgiving pointer invalidity.

### A `make_ptr_prospective()` Function

This section describes a convenience mechanism for encapsulating the pair of `reinterpret_cast<>` operations used to create a prospective pointer value as described in the “Consequences for pointer zap” section of P2434R1.

A `make_ptr_prospective()` function takes a pointer as its argument and returns the corresponding prospective pointer value. This function can be based on a `reinterpret_cast` pair as suggested in P2434R1 (see the “Consequences for pointer zap” section), for example, by using an `uintptr_t` data member private to `usable_ptr<T>`. As suggested by the “Proposal” section of that same paper, the `memcpy()` function could instead be used.

### A `usable_ptr<T>` Template Class

This section describes a convenience mechanism for encapsulating the pair of `reinterpret_cast<>` operations used to create a prospective pointer value as described in the “Consequences for pointer zap” section of P2434R1.

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. This template class can be based on a `reinterpret_cast` pair as suggested in P2434R1 (see the “Consequences for pointer zap” section), for example, by using an `uintptr_t` data member private to `usable_ptr<T>`. As suggested by the “Proposal” section of that same paper, the `memcpy()` function could instead be used.

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

### Atomic Operations Forgive Pointer Invalidity

This section describes a convenience mechanism in which `std::atomic<T*>` has special behavior so that the pointer values become prospective, as alluded to in the “Consequences for pointer zap” section of P2434R1.

Atomic operations have the side effect of producing prospective pointer values, thus 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.  Note that this means that an implementation could choose to define `usable_ptr<T>` in terms of `atomic<T*>`.

Although this is an attractive approach, it conflicts with a desire to treat all atomic operations as `constexpr`.  We therefore ask that atomic operations have the side effect of producing prospective pointer values without specifying the mechanism, thus avoiding this conflict.  One possible

Although concerns were raised at the 2022 Kona meeting about possible optimization limitations from this approach, the fact is that any thread might update a given atomic pointer 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), that pointer value becomes a prospective pointer value.

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

This section describes a convenience mechanism in which `volatile` operations have special behavior so that any associated pointer values become prospective.

This means that `volatile` operations on pointers have the side effect of producing prospective pointer values, thus forgiving pointer invalidity.  One way to think of this is that the operations accessing such objects use `reinterpret_cast<>` operations, as described in the “Consequences for pointer zap” section of P2434R1.

We believe this to be de facto status quo given the current semantics required of volatile by real-world device drivers.

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++).  In other words, the value loaded from a volatile pointer might have no relation to the value most recently stored to that same pointer, and all loads and stores are by C or C++ code.

# Examples

## LIFO Push

A simple (but according to the standard, buggy) 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 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 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 top_.exchange(nullptr); }
};
```

Alert readers will notice that there is no yellow code, that is, there are no code changes required.

The first use case is fixed in part by the convenience behaviors of std::atomic<T*>, which cause pointers loaded from atomics to become provisional. Also required are the additional changes in P3347R0 ("Invalid/Prospective Pointer Operations"), which proposes that pointer invalidity not modify representation bytes and also due to the advent of prospective pointer values:

- 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 prospective pointer `A` in `.compare_exchange_weak`. However, this atomic operation looks only at representation bytes, which must not be unaffected by the fact that the pointer value is prospective, which fixes this example. If the `.compare_exchange_weak` operation fails, this prospective pointer will remain unused, so no harm is done. Execution will proceed with the new value provided by that failing `.compare_exchange_weak` operation.

The second use case is fixed in the same way:

- `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`, which is a prospective pointer value due to the `make_ptr_provisional()`
- 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 prospective pointer value
- 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 prospective pointer value. However, the referenced object has now been fully constructed, so that this prospective pointer value is now a valid pointer to the new object. Note that `X3` has been exposed by virtue of being pushed onto the stack, and having been stored in the atomic object `top_`. Had `X3` not been exposed, the implementation would have been under no obligation to use its provenance.

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;
```

This will re-evaluate provenance on `p` according to its representation bytes any time that `p` would otherwise be an invalid pointer.

## Wording

The following sections describe adding a `usable_ptr<T>` class and a `make_ptr_prospective()` function to the `<memory>` header.

### Add usable_ptr<T> and make_ptr_prospective()

**n.m Class usable_ptr** **[usable.ptr]**

**n.m.1 General** **[usable.ptr.general]**

```
namespace std {
  template <typename T>
  class usable_ptr {
    uintptr_t iptr; // exposition only
  public:

    // n.m.2, member functions
    usable_ptr(T* ptr) noexcept;
    T operator*() const noexcept;
    operator T*() const noexcept;
  };

  // n.m.3, non-member functions
  template<class T> T* make_ptr_prospective(T* ptr) noexcept;
}
```

**n.m.2 Member functions** **[usable.ptr.members]**

**`usable_ptr(T* ptr = nullptr) noexcept;`**

*Effects*: Initializes `iptr` with `reinterpret_cast<uintptr_t>(ptr)`.

**`T operator*() const noexcept;`**

*Returns*: `*get()`.

**`operator T*() const noexcept;`**

*Returns*: `get()`.

**`T* operator->() const noexcept;`**

*Returns*: `get()`.

**`T* get() const noexcept;`**

*Returns*: `reinterpret_cast<T*>(iptr)`, which is a pointer with prospective provenance.

**`T* operator=(T* ptr) noexcept;`**

*Effects*: Assigns `reinterpret_cast<uintptr_t>(ptr)` to `iptr`.

*Returns*: `get()` using the new value of `iptr`.

**n.m.3 Non-member functions [usable.ptr.func]**

**`template<class T> T* make_ptr_prospective(T* ptr) noexcept;`**

*Returns*: If `ptr` is a null pointer, `nullptr`. Otherwise,
`reinterpret_cast<T*>(reinterpret_cast<uintptr_t>(ptr))`, which is a prospective pointer whose value bits match those of `ptr`.

## Atomic Operations And Prospective Pointers

### # [atomics.types.generic.general]

Add after paragraph 3:

The specialzation `atomic<T*>` has special conversion semantics, so that given an object `x` of type `atomic<T*>`, a conversion from `x` to a compatible type `U*` behaves as if it were written:

```
reinterpret_cast<U*>(reinterpret_cast<uintptr_t>(x))
```

Similarly, given an object `y` of type `T*`, a conversion from `y` to a compatible type `atomic<U*>` behaves as if it were written:

```
reinterpret_cast<atomic<U*>>(reinterpret_cast<uintptr_t>(y))
```

## Volatile Operations And Prospective Pointers

### # [intro.abstract]

Change paragraph 7.1 to read:

Accesses through volatile glvalues are evaluated strictly according to the rules of the abstract machine, however, conversions from an object `x` of type `T* volatile` to a compatible type `U*` behave as if they were written

```
reinterpret_cast<U*>(reinterpret_cast<uintptr_t>(x))
```

Similarly, conversions from an object `y` of type `T*` to a compatible type `U* volatile` behave as if they were written:

```
reinterpret_cast<U* volatile>(reinterpret_cast<uintptr_t>(y))
```

## History

P2414R4:

- Updated based on the June 24, 2024 St. Louis SG1 review:
  - Fix numerous typos.
  - Drop discussion of defining load, store, and arithmetic operations on invalid and prospective pointers to allow them to be in their own paper.
  - Add function as well as class.
- Added draft wording and updated per Daniel Krügler feedback.
- Move the history section to the end of the paper.

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 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.

## 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 P2434R1

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 builds on top of that paper in order to provide ways for the user to avoid pointer zap.