ISO/IEC JTC1/SC22/WG21 P2414R1, 2021-08-13

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, 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 class template std::usable_ptr<T> that is a pointer-like type that is still usable after its pointee's lifetime has ended; and (2) that atomic and volatile operations be redefined to forgive lifetime-end pointer invalidity. For more details, see options A2, A3, B2, B3, and B5).

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 to cause all pointers to be exempt from lifetime-end pointer invalidity, for example, a command-line option.

History

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 (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, as a convenience to the user, this paper proposes that atomic operations have the side effect of forgiving pointer invalidity (details in sections A3, B3, and B5).

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.

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.

Target Users and Use Cases

Different users have different invalid-pointer needs:

1. Those writing new code are likely to be able to use whatever facilities the standard and their vendor provide to identify zap-susceptible algorithms, as will those working on existing systems where it is feasible to identify and fix zap-susceptible algorithms.
2. Those working on existing systems with legacy guru-free code might nevertheless be able to identify some attributes of zap-susceptible algorithms. For example, the types of all variables involved in such algorithms might be known, but it might not be feasible to identify these algorithms on an access-by-access basis. Such developers might find it feasible to mark types (or allocations and frees involving instances of those types), but might not be able to carry out marking on the more-efficient access-by-access basis, despite the likelihood that access-by-access marking would give the compiler more freedom to optimize.
3. Those working on large existing systems with legacy guru-free code might need to take the conservative approach of using a compiler flag to implicitly mark all accesses (or all allocations and frees) as susceptible to pointer invalidation. This might be an intermediate state while source-code markings were being applied.
4. Those integrating large numbers of existing projects might not be in a position to modify each and every build system, let alone apply source-code markings, especially in cases where the source code is not available. Such people might quite understandably demand that all pointers always be treated as “bags of bits”. Other than the somewhat controversial bag of bits, we do not yet have a good solution for this group of users, but we hope to ease their predicaments at least somewhat by working to ensure that the problem of pointer zap and its various solutions are well-known, hopefully increasing the probability that the developers of each project takes appropriate invalidation actions from the earlier items on this list.
5. Those using third-party proprietary libraries cannot build those libraries or inspect those libraries’ source code. These people would like to use a global setting (for example, a compiler flag) to implicitly mark everything, that is, allocations, accesses, *and* pointers. They also need to know what that means in terms of interactions with the library code as well as under what conditions such a global setting will successfully avoid pointer-zap issues.
6. Your pointer-invalidation needs here!

One specific example of a trap that might await those using third-proprietary libraries is a library interface that expects a pointer to a callback, but through which one can also pass an integer that was obtained from a pointer via a cast operation. The library might cast the integer to `void *` then cast it back to the original integer type within the callback. The user might well hope that freeing of an object whose address happened to match the integer should not cause the callback to get a different integer than the one that was passed in through the interface. (Yes, template metaprogramming can often avoid the need for this, however, type erasure really is needed from time to time even in C++.)

Different algorithms also have pointer-zap needs:

- Type of access to invalid pointers:
    - Access invalid pointers, but never dereference them.
    - Access invalid pointers, and also dereference zombie pointers.
      Note: No valid algorithm ever dereferences non-zombie invalid pointers.
- Type of memory in use:

- Only dynamic-lifetime storage. Marking allocations and frees works in this case, and will be the simplest and most robust approach in many cases. However, marking accesses and/or pointer objects gives the compiler more freedom to optimize.
- On-stack storage, although perhaps also using dynamic-lifetime storage. This case requires accesses and/or pointer objects be marked. Marking allocations and frees is insufficient.

Options for Invalid Pointer Use

A1: Allow invalid pointer use
A2: Marking of invalid pointers to allow use
A3: Allow use of invalid pointers with pre-existing markings
A4: Allow use of invalid pointers to objects with marked allocations and/or deallocations

These options are not necessarily mutually exclusive, so that a successful approach might combine several of them.

A1: Allow invalid pointer use

This option forbids the compiler from distinguishing between valid and invalid pointers, and can thus be said to treat pointers as bags of bits.

A2: Marking of invalid pointers as usable

This option does not change the treatment invalid pointers in general, but allows the user to mark specific pointer and/or pointer accesses as special. When a pointer or access is so marked, the compiler is not permitted to use any prior lifetime-related provenance, but instead must recompute any such provenance from the representation bytes, type, and so on. Here is a prototype `usable_ptr` class that is intended for such marking:

```
template <typename T> class usable_ptr {
 public:
  // Showing only essential member functions
  constexpr usable_ptr(T* p = nullptr) noexcept;
  constexpr T* get() const noexcept;
  constexpr bool operator==(usable_ptr<T>& o) const noexcept;
};
template <typename T> constexpr usable_ptr<T>& usable_ref(T*& p) noexcept;
template <typename T>
constexpr atomic<usable_ptr<T>>& atomic_usable_ref(atomic<T*>& p) noexcept;
```

LIFO-list marking of invalid pointers

The following code uses `usable_ptr` to mark the zap-tolerant LIFO list algorithm:

```
template <typename T> class LIFOList { // T must have accessible T* next_
  std::atomic<T*> top_{nullptr};
```

```
 public:
  void push(T* p) {
    atomic<usable_ptr<T>>& reftop = usable_atomic_ref(top_);
    usable_ptr<T>& refnext = usable_ref(p->next_);
    refnext = reftop.load(); // Pointer may become invalid before CAS
    while (!reftop.compare_exchange_weak(refnext, usable_ptr<T>(p))) {}

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

Here is how these changes handle the use of invalid pointers:

- Line 1 of `push`: Creating an `atomic<usable_ptr<T>>` reference `reftop` of `top_` so that the invalidity of values loaded from `top_` (represented by `reftop`) is forgiven. Note that the values in `top_` are always valid, but they may become invalid right after being loaded.
- Line 2 of `push`: Creating a `usable_ptr` reference `refnext` of `p->next_` to indicate that the invalidity of `p->next_` is forgiven as long as `refnext` is in scope.
- Line 3 of `push`: Writing an invalid pointer into `refnext` representing `p->next_`. If the pointer became invalid right after being loaded from `top_` (represented by `reftop`) and before being written to `p->next_`, then this invalidity is forgiven by the existence of `refnext`.
- Line 4 of `push`: Passing an invalid pointer (`p->next_` represented by `refnext`) to `compare_exchange`. This invalidity would be forgiven by the existence of `refnext`.
- Line 4 of `push`, `compare_exchange`: Reading a usable invalid pointer from `p->next_` (represented by `refnext`) and comparing with it.
- Line 4 of `push`, failed `compare_exchange`: Writing the value found in `top_` (represented by `reftop`) into `p->next_` (represented by `refnext`). The value may have just become invalid after being read from `top_`. The invalidity is forgiven by loading from `reftop` and the existence of `refnext`.

Ths godbolt illustrates this code using the `address_space(42)` attribute approach to defend against pointer invalidation.

Hazard-pointer marking of invalid pointers

The following code uses `usable_ptr` to mark the user-code portion of a simple hazard-pointers use case:

```
void reader(std::atomic<T*>& src) {
  usable_ptr<T> ptr = atomic_usable_ref(src).load();
  if (hazard_pointer::try_protect(ptr, src))
    f(ptr.get()); // ptr.get() is guaranteed to be valid
  hazard_pointer::clear();
}
```

The following code uses `usable_ptr` to mark a simplified hazard-pointers library implementation:

```
std::atomic<usable_ptr<T>> _hp;
bool try_protect(usable_ptr<T>& ptr, std::atomic<T*>& src) {
```

```
  usable_ptr<T> p = ptr;
  _hp.store(ptr);
  ptr = atomic_usable_ref(src).load();
  return ptr == p;
}
void clear() { _hp.store(usable_ptr<T>(nullptr)); }
```

Here is how these changes handle the use of invalid pointers:

- Line 1 of `reader`: Writing an invalid pointer into `ptr`. The pointer may have just become invalid after being loaded. This invalidity would be forgiven by the `usable_ptr<T>` nature of `ptr` and by loading from `atomic_usable_ref(src)`.
- Line 2 of `reader`: Passing an invalid pointer `ptr` to `hazard_pointer::try_protect`. This invalidity is forgiven by the `usable_ptr<T>` nature of `ptr`.
- Line 3 of `reader`: Passing an invalid pointer to `f`. This invalidity is forgiven by the `usable_ptr<T>` nature of `ptr`.
- Line 1 of `try_protect`: Read an invalid pointer from `ptr` and store it into `p`. This invalidity is forgiven by the `usable_ptr<T>` nature of `ptr` and `p`.
- Line 2 of `try_protect`: Writing an invalid pointer into `_hp`. The invalidity would be forgiven by the `usable_ptr` nature of `_hp`.
- Line 3 of `try_protect`: Writing an invalid pointer into `ptr`. The pointer may have just become invalid after being loaded. This invalidity would be forgiven by the `usable_ptr<T>` nature of `ptr` and loading from `atomic_usable_ref(src)`.
- Line 4 of `try_protect`: Reading and comparing invalid pointers from `ptr` and/or `p`. This invalidity is forgiven by the `usable_ptr<T>` nature of `ptr` and `p`.

A3: Allow use of invalid pointers with pre-existing markings

Please note that this is not a complete solution, but it does improve ease of use when used in conjunction with other options.

This option provides a number of existing library functions with usable_ptr semantics. These library functions include atomics and volatile accesses, plus outside-of-standard facilities such as inline assembly. This approach leverages the principle of least surprise: Atomics are used in situations where pointer invalidation is problematic and where determining provenance is difficult, thus providing great benefit to those writing zap-tolerant concurrent algorithms at little or no cost in terms of optimization.

The following code relies on naturally occurring C++ atomics to mark the `top_` pointer used by the zap-tolerant LIFO list algorithm and employs `usable_ptr<T>` as needed for other objects:

```
template <typename T> class LIFOList { // T must have accessible T* next_
  std::atomic<T*> top_{nullptr};
 public:
  void push(T* p) {
    usable_ref(p->next_) = top_.load();
```

```
    while (!top_.compare_exchange_weak(refnext_, p)) {}
  }
  T* pop_all() { return top_.exchange(nullptr); }
}; // LIFOList
```

Here is how these changes handle the use of invalid pointers:

- Line 1 of `push`: Writing an invalid pointer into `usable_ref(p->next_)`. If the pointer became invalid right after being loaded from `top_` and before being written to `p->next_`, then this invalidity would be forgiven by the `usable_ptr<T>::usable_ref`.
- Line 2 of `push`: Passing an invalid pointer (`p->next_`) to `compare_exchange`. This invalidity would be forgiven by the atomic nature of `compare_exchange`.
- Line 2 of `push`, `compare_exchange`: Reading invalid pointer from `p->next_` and comparing with it. The atomic nature of `compare_exchange` forgives the invalidity.
- Line 2 of `push`, failed `compare_exchange`: Writing the value found in `top_` into `p->next_`. The value may have just become invalid after being read from `top_`. The atomic nature of `compare_exchange` forgives the invalidity.

The following code uses C++ atomics to mark the user-code portion of a simple hazard-pointers use case:

```
void reader(std::atomic<T*>& src) {
  usable_ptr<T> ptr = src.load();
  if (hazard_pointer::try_protect(ptr, src))
    f(ptr.get()); // ptr.get() is guaranteed to be valid
  hazard_pointer::clear();
}
```

The following code uses C++ atomics to mark a simplified hazard-pointers library implementation, but note that this approach can result in type issues and suboptimal code in many cases:

```
std::atomic<T*> _hp;
bool try_protect(usable_ptr<T>& ptr, std::atomic<T*>& src) {
  usable_ptr<T> p = ptr;
  _hp.store(ptr);
  ptr = src.load();
  return ptr == p;
}
void clear() { _hp.store(nullptr); }
```

Here is how these changes handle the use of invalid pointers:

- Line 1 of `reader`: Writing an invalid pointer into `ptr`. The pointer may have just become invalid after being loaded. This invalidity would be forgiven by the `usable_ptr<T>` nature of `ptr` and the atomic nature of `src`.
- Line 2 of `reader`: Passing an invalid pointer `ptr` to `hazard_pointer::try_protect`. This invalidity is forgiven by the `usable_ptr<T>` nature of `ptr`.
- Line 3 of `reader`: Passing an invalid pointer to `f`. This invalidity is forgiven by the `usable_ptr<T>` nature of `ptr`.

- Line 1 of `try_protect`: Read an invalid pointer from `ptr` and write it into `p`. This invalidity is forgiven by the `usable_ptr<T>` nature of `ptr` and `p`.
- Line 2 of `try_protect`: Writing an invalid pointer into `_hp`. Invalidity would be forgiven because this is an atomic store.
- Line 3 of `try_protect`: Writing an invalid pointer into `ptr`. The pointer may have just become invalid after being loaded. This invalidity would be forgiven by the `usable_ptr<T>` nature of `ptr` and the atomic nature of `src`.
- Line 4 of `try_protect`: Reading invalid pointers from `ptr` and/or `p`. This invalidity is forgiven by the `usable_ptr<T>` nature of `ptr` and `p`.

Note that even in current implementations, volatile objects must presume wildcard provenance for invalid pointers because doing otherwise would break device drivers for devices making use of virtual addresses (for example, devices behind I/O MMUs). Allowing the use of invalid pointers for volatile and atomic operations leverages the principle of least surprise. However, using volatile and atomic operations when not needed incurs performance and type-system side effects that are unacceptable in many situations.

A4: Allow use of invalid pointers to objects with marked allocations and/or deallocations

The idea here is to codify the long-standing practice of hiding allocations and frees from the compiler. Such hiding has traditionally been accomplished by coding an allocator as part of the application, but using names that the compiler does not recognize. Of course, one issue with this long-standing approach is that it rules out use of library functions that use the C++ `new` and `delete` operators. The goal of this section is to permit use of `new` and `delete` while also allowing the user to avoid pointer-invalidation issues.

Neither of the current examples allows easy demonstration of this approach, so let's focus on an allocation:

```
auto p = new T;
```

This allocation could be hidden by a command-line parameter, through use of a memory allocator that is unknown to the compiler, or by marking it:

```
auto p = usable_ptr(new T);
```

Similarly, deallocation might be hidden as follows:

```
delete usable_ptr(p);
```

Either way, the compiler is forbidden from invalidating any pointers based on either operation.

A user of the LIFO push library could then write code as follows:

```
auto p = usable_ptr(new my_type);
push(p);
p = pop_all();
```

```
while (p) {
   q = p->next_;
   foo(p);
   delete usable_ptr(p);
   p = q;
}
```

Because the `usable_ptr()` invocations hide the allocations and frees from the compiler, the LIFO push library is insulated from pointer-invalidation issues. The advantage of marking the pointers and accesses within the LIFO push library is that the user is completely insulated from any pointer-invalidation issues. However, in large legacy guru-free code bases, and especially in cases where the library's source code is not available, marking allocations and deallocations can be much more straightforward than marking pointers and accesses.

Options for Zombie Pointer Dereference

B1: Allow zombie pointer dereference
B2: Marking of zombie pointers to allow dereference
B3: Allow dereference of zombie pointers with pre-existing markings
B4: Allow dereference of zombie pointers to objects with marked allocations and/or deallocations
B5: Successful compare_exchange overwrites expected value

These options are not necessarily mutually exclusive, so that a successful strategy might combine some of them.

B1: Allow zombie pointer dereference

This option forbids the compiler from distinguishing between valid and invalid pointers, and can thus be said to treat pointers as bags of bits. This approach preserves legacy code, but requires significant implementation changes that might limit optimization, and also requires modest changes to the standard.

B2: Marking of zombie pointers to allow dereference

This approach is similar to that described in A2 ("Marking of invalid pointers to allow use"), but also permits dereferencing of marked zombie pointers. It is also the approach proposed in the P1726R5 section entitled "Limit Pointer Invalidation via Marking of Dereferenced Pointer Fetches". This approach requires changes to legacy code, to implementations, and to the standard.

B3: Allow dereference of zombie pointers with pre-existing markings

Please note that this is not a complete solution, but it does improve ease of use when used in conjunction with other options.

This approach is similar to that described in A3 ("Allow use of invalid pointers with pre-existing markings"), but also permits dereferencing of zombie pointers that have pre-existing markings or that were fetched using marked accesses. It is also the approach proposed in the P1726R5 section entitled "Limit Lifetime-End Pointer Zap Based on Marking of Pointers and Fetches". This approach requires changes to legacy code, to implementations, and to the standard. This

approach reduces the number of changes required to legacy code. It might require small changes to implementations to take into account the extra semantics now assigned to atomics, and it will also require corresponding changes to the standard.

Here is how these changes handle the dereferencing of zombie pointers:

- If `pop_all` returns a pointer to the first node in a linked list with more than one node, then nonnull `next_` pointers may be zombies.
- The caller of `pop_all()` must apply `usable_ptr<T>` to these nonnull `next_` pointers.
- Of course, if the pointers were non-zombie invalid pointers, the dereference operation would still invoke undefined behavior.

As noted in A3, that even in current implementations, volatile objects must presume wildcard provenance for invalid pointers because doing otherwise would break device drivers for devices making use of virtual addresses (for example, devices behind I/O MMUs). Allowing the use of invalid pointers for volatile and atomic operations leverages the principle of least surprise. However, using volatile and atomic operations when not needed incurs performance and type-system side effects that are unacceptable in many situations.

B4: Allow dereference of zombie pointers to objects with marked allocations and/or deallocations

This approach is similar to that described in A4 (“Allow use of invalid pointers to objects with marked allocations and/or deallocations”), but also permits dereferencing of marked zombie pointers. It is also the approach proposed in the P1726R5 section entitled “Mark Allocations and Deallocations (Zombie Only)”. This approach requires changes to legacy code, to implementations, and to the standard. An outside-of-standard introduction of a command-line flag that caused all allocations and deallocations to be treated as if they were marked would greatly reduce required changes to legacy code, following long-standing practice of hiding allocators from the compiler.

B5: Overwrite zombie pointers with valid pointers explicitly or conceptually

In the benign ABA pattern, zombie pointer dereferences arise following successful comparisons of zombie pointers with the corresponding valid pointers. In our proposed solution, the zombie pointers are overwritten with the corresponding valid pointers upon the success of the comparison and before being dereferenced.

Overwriting zombie pointers can take one of two forms:

1. Explicit overwrite by the programmer or
2. Conceptual overwrite by a successful compare_exchange (CAS) operation of the expected pointer value with the value read by that CAS operation from the atomic pointer.

Explicit overwrite

If the comparison is done not as a part of a CAS operation, then it is the programmer’s responsibility to ensure that the potentially zombie pointer is overwritten by the corresponding valid pointer before dereferencing it.

For example, consider the following implementation of hazard pointer `try_protect`. Note the difference from the implementation in A2 and A3 that makes the following code susceptible to zombie dereference because here `try_protect` might return `true` when `ptr` is a zombie:

```
bool try_protect(T*& ptr, std::atomic<T*>& src) {
    _hp.store(ptr);
    T* p = src.load();
    if (ptr == p) return true; // ptr may be a zombie
    ptr = p;
    return false;
}
```

This proposed solution would require programmers to write code in a way that guarantees that zombie pointers are overwritten by the corresponding valid pointers before being dereferenced, thus eliminating the zombie pointer dereference in such cases.

For example, the above code can be rewritten as follows to prevent zombie dereference by overwriting `ptr` in case of success (as well as failure):

```
bool try_protect(T*& ptr, std::atomic<T*>& src) {
    _hp.store(ptr);
    T* p = src.load();
    bool result = (ptr == p);
    ptr = p; // Overwrite ptr with valid pointer in case of success
    return result;
}
```

Or, alternatively, the code can be written similar to the examples under A2 and A3:

```
bool try_protect(T*& ptr, std::atomic<T*>& src) {
    T* p = ptr;
    _hp.store(p);
    ptr = src.load(); // Overwrite ptr with valid pointer in case of success
    return ptr == p;
}
```

The above code examples avoid zombie pointer dereference but still need a solution for invalid pointer use.

Conceptual overwrite by successful CAS

When the successful comparison that leads to zombie dereference is part of a CAS operation, the programmer has no opportunity to explicitly overwrite the zombie pointer after the success of CAS because of the atomicity of the successful comparison with the writing to the target of CAS. A different thread might read the target immediately after the success of CAS and then reach the zombie pointer and dereference it. Therefore we need to make the overwrite implicitly part of the CAS.

Currently, a successful CAS operation leaves the expected value untouched. This is only reasonable, given that the expected value is equal to the actual value. The point of this alternative is not to force useless writes following successful CAS operations, but rather to force the compiler to treat the zombie pointer just as if it is the same as the valid pointer that is read from the target of CAS. Therefore, if the expected value was a zombie pointer, it is no longer considered to be a zombie. In particular, later access to and dereferencing of this pointer will proceed normally as if it were the corresponding valid pointer.

Note that if the pointer is not a zombie pointer, but is instead an invalid pointer to freed memory that has not be reallocated, or that has been reallocated as an object of an incompatible type, then dereferencing the pointer still invokes undefined behavior.

With this proposed change to the semantics of successful CAS, the LIFOList example in A2 becomes automatically compliant for zombie dereference.

The code however cannot be written as follows because p->next_ needs to be modeled as overwritten by the successful CAS:

```
    void push(T* p) {
      while (true) {
        T* t = top_.load();
        p->next_ = t;
        if (top_.compare_exchange_weak(t, p)) return;
      }
    }
```

Effect of this solution:

- Code changes: Some. Ensure that the zombie pointer is always overwritten before dereference by the valid pointer that it compared equal to.
- Implementation changes: None or small.
- Standard changes: Simple. Change the semantics of successful CAS as if the expected value is overwritten by the value read from the target.

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 A: Implementation and Use of `usable_ptr<T>`

https://godbolt.org/z/rn9dxoroa

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

///
/// LIBRARY CODE -- Essential components only.
///

// Template to mark pointers that may become invalid
template <typename T> class usable_ptr {
  intptr_t _intp;
 public:
  constexpr usable_ptr(T* p = nullptr) noexcept : _intp(reinterpret_cast<intptr_t>(p)) {}
  constexpr T* get() const noexcept { return reinterpret_cast<T*>(_intp); }
  constexpr bool operator==(const usable_ptr<T>& o) const noexcept { return _intp == o._intp; }
}; // usable_ptr

template <typename T> constexpr usable_ptr<T>& usable_ref(T*& p) noexcept {
  return reinterpret_cast<usable_ptr<T>&>(p);
}

template <typename T>
constexpr std::atomic<usable_ptr<T>>& atomic_usable_ref(std::atomic<T*>& src) noexcept {
  return reinterpret_cast<std::atomic<usable_ptr<T>>&>(src);
}

////////////////////////////////////////////////////////////////////////////////

///
/// USER CODE
///

// Programmer 1: Implements LIFOList using usable_ptr.
```

```
/// Comments include code with unmarked pointers
template <typename T> class LIFOList { // T must have accessible T* next_
  std::atomic<T*> top_{nullptr};
 public:
  void push(T* p) {
    /// p->next_ = top_.load(); // Pointer may become invalid before CAS
    /// while (!top_.compare_exchange_weak(p->next_, p)) {}
    std::atomic<usable_ptr<T>>& reftop = atomic_usable_ref(top_);
    usable_ptr<T>& refnext = usable_ref(p->next_);
    refnext = reftop.load(); // Pointer may become invalid before CAS
    while (!reftop.compare_exchange_weak(refnext, usable_ptr<T>(p))) {}
  }
  T* pop_all() { return top_.exchange(nullptr); }
}; // LIFOList

// Programmer 2: Doesn't know about LIFOList or usable_ptr.
struct Node { Node* next_; };

// Programmer 3: Knows about Node but doesn't know about LIFOList and usable_ptr.
void consume_list(Node* p) {
  while (p) {
    std::cout << p << " ";
    // p may be a zombie pointer (starting from the second iteration).
    Node* next = p->next_;
    delete p;
    p = next;
  }
  std::cout << std::endl;
}

// Programmer 4: Knows about Node and LIFOList but doesn't know about usable_ptr.

LIFOList<Node> l;

Node* construct_list() {
  for (int i = 0; i < 5; ++i) {
    Node* p = new Node;
    std::cout << p << " ";
    l.push(p);
  }
  std::cout << std::endl;
  return l.pop_all();
}

// Programmer 5: Knows only about construct_list and consume_list
int main() {
```

```
  consume_list(construct_list());
}
```