ISO/IEC JTC1/SC22/WG21 D1726R2, 2019-11-05

Pointer lifetime-end zap

Authors: Paul E. McKenney, Maged Michael, Jens Mauer, Peter Sewell, Martin Uecker, Hans Boehm, Hubert Tong, Niall Douglas, Thomas Rodgers, Will Deacon, and Michael Wong
Other contributors: Martin Sebor, Florian Weimer, Davis Herring, Rajan Bhakta, David Goldblatt, Hal Finkel, Kostya Serebryany, and Lisa Lippincott.

Abstract

The C++ standard currently specifies that all pointers to an object become invalid at the end of its lifetime. This *lifetime-end pointer zap semantics* permits some additional diagnostics and optimizations, some deployed and some hypothetical, but 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. This paper presents one of these algorithms and discusses some possible resolutions, ranging from retaining the status quo to completely eliminating lifetime-end pointer zap.

Note that some of these algorithms also have issues with current pointer-provenance wording, however, these provenance issues will be addressed in separate papers.

History

P1726R1 was presented at Belfast in 2019.

- Added a first draft of wording changes.
- Added reference to related provenance issues.

P1726R0 was presented at Koeln in 2019.

- Added analysis of the SPARC ADI feature and the ARMv8 MTE feature, each of which enable trapping on dereferencing of invalid pointers and each of which is completely compatible with the elimination of lifetime-end pointer zap.
- Added a detailed sequence of events showing how SPARC ADI and ARMv8 MTE would interact with the LIFO singly linked push algorithm, showing both detection of an invalid pointer and a false-negative event where an invalid pointer remained undetected. (LIFO singly linked list push operates properly in both cases.)
- Added a possible resolution involving zapping only those pointers that are actually passed to `delete` and similar.
- Added a possible resolution involving converting all pointers to integers.
- Added an informal evaluation of possible resolutions carried out at CPPCON 2019.

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 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 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.

What Does the C++ Standard Say?

This section refers to Working Draft N4800.

6.6.5 Storage duration [basic.stc], paragraph 4 reads as follows:

> *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 (6.7.2). Indirection through an invalid pointer value and passing an invalid pointer value to a deallocation function have undefined behavior. Any other use of an invalid pointer value has implementation-defined behavior. [34]*
>
> *[34] Some implementations might define that copying an invalid pointer value causes a system-generated runtime fault.*

This clearly indicates that as soon as an object's lifetime ends, all pointers to it instantaneously become invalid, and use of such pointers has implementation-defined behavior.

6.6.5.4.3 Safely-derived pointers [basic.life], paragraph 3 reads as follows:

> *An integer value is an integer representation of a safely-derived pointer only if its type is at least as large as std::intptr_t and it is one of the following:*
>
> *(3.1) - the result of a reinterpret_cast of a safely-derived pointer value;*
> *(3.2) - the result of a valid conversion of an integer representation of a safely-derived pointer value;*

(3.3) - the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained an integer representation of a safely-derived pointer value;
(3.4) - the result of an additive or bitwise operation, one of whose operands is an integer representation of a safely-derived pointer value P, if that result converted by reinterpret_cast<void> would compare equal to a safely-derived pointer computable from reinterpret_cast<void*>(P).*

And paragraph 4 reads as follows:

> *An implementation may have relaxed pointer safety, in which case the validity of a pointer value does not depend on whether it is a safely-derived pointer value. Alternatively, an implementation may have strict pointer safety, in which case a pointer value referring to an object with dynamic storage duration that is not a safely-derived pointer value is an invalid pointer value unless the referenced complete object has previously been declared reachable (19.10.5). [Note: The effect of using an invalid pointer value (including passing it to a deallocation function) is undefined, see 6.6.5. This is true even if the unsafely-derived pointer value might compare equal to some safely-derived pointer value. — end note] It is implementation-defined whether an implementation has relaxed or strict pointer safety.*

This reiterates that invalid pointers remain invalid, but the combination of these two paragraphs allows a pointer value to be sequestered in a suitably large integer (but only if this sequestration occurs while the pointer is still valid) and then converted back to a pointer after the storage has been reallocated.

6.7.2 Compound types [basic.compound], bulleted paragraph 3:

(3.1) - a pointer to an object or function (the pointer is said to point to the object or function), or
(3.2) - a pointer past the end of an object (7.6.6), or
(3.3) - the null pointer value (7.3.11) for that type, or
(3.4) - an invalid pointer value.

Note that a pointer to an object that is freed and later to an object at that same address has only the option of being an invalid pointer value.

6.8.3 Object and reference lifetime [basic.life], paragraph 6:

> *Before the lifetime of an object has started but after the storage which the object will occupy has been allocated [32] or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that represents the address of the storage location where the object will be or was located may be used but only in limited ways. For an object under construction or destruction, see 10.9.4. Otherwise, such a pointer refers to allocated storage (6.6.5.4.1), and using the pointer as if the pointer were of type void*, is well-defined. Indirection through such a pointer is permitted but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:*
>
> *(6.1) - the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a delete-expression,*
> *(6.2) - the pointer is used to access a non-static data member or call a non-static member function of the object, or*
> *(6.3) - the pointer is implicitly converted (7.3.11) to a pointer to a virtual base class, or*

(6.4) - the pointer is used as the operand of a static_cast (7.6.1.8), except when the conversion is to pointer to cv void, or to pointer to cv void and subsequently to pointer to cv char, cv unsigned char, or cv std::byte (16.2.1), or
(6.5) - the pointer is used as the operand of a dynamic_cast (7.6.1.6).

[32] For example, before the construction of a global object that is initialized via a user-provided constructor (10.9.4).

6.8.3 Object and reference lifetime [basic.life], paragraph 7:

Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. For an object under construction or destruction, see 10.9.4. Otherwise, such a glvalue refers to allocated storage (6.6.5.4.1), and using the properties of the glvalue that do not depend on its value is well-defined. The program has undefined behavior if:

(7.1) - the glvalue is used to access the object, or
(7.2) - the glvalue is used to call a non-static member function of the object, or
(7.3) - the glvalue is bound to a reference to a virtual base class (9.3.3), or
(7.4) - the glvalue is used as the operand of a dynamic_cast (7.6.1.6) or as the operand of typeid.

6.8.3 Object and reference lifetime [basic.life], paragraph 8:

If, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object, if:

(8.1) - the storage for the new object exactly overlays the storage location which the original object occupied, and
(8.2) - the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
(8.3) - the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and
(8.4) - neither the original object nor the new object is a potentially-overlapping subobject (6.6.2).

These three paragraphs allow pointers to be reused, but only if the underlying storage has remained allocated throughout. This might help in some situations, but not for ABA-tolerant concurrent algorithms. For example, as noted earlier, relaxed pointer safety does not extend to permitting predictable use of invalid pointers. Therefore, the C++ standard really does allow implementations to zap any and all pointers to any object whose lifetime ends. And this really does outlaw important and long-standing concurrent algorithms.

Pointers have representation bytes. Pointers to different objects of non-zero size with overlapping lifetimes cannot have equal representation bytes, but pointers to different non-zero-sized objects (as opposed to subobject) having non-overlapping lifetimes might well compare equal.

K&R (first edition) appears not to say anything analogous about pointers to lifetime-ended objects. Invalid pointers of this sort thus appears to be a more recent invention.

Rationale for Lifetime-End Pointer Zap Semantics

There are several motivations one might have for the lifetime-end pointer zap semantics, some current, some hypothetical, and some historic.

Diagnose, or Limit Damage From, Use-After-Free Bugs

As far as we can determine, the most substantial current motivation for lifetime-end pointer zap is to limit damage from use-after-free bugs, especially in cases where the address of an automatic-storage-duration variable is taken but then mistakenly returned.

Martin Uecker noted that some compilers will unconditionally return `nullptr` in cases like this:

```
extern void* foo(void) {
        int aa;
        void* a = &aa;
        return a;
}
```

If this is a bug, and the return value is used for a load or store, returning `NULL` will make the bug easier to find than returning a pointer containing the bits that used to reference `aa`. However, as Hans Boehm noted, issuing a diagnostic would be even more friendly, and compilers can and do emit warnings in such cases, so this argument only really applies for codebases compiled without warnings.

Florian Weimer adds that manually invalidating a pointer after a call to `free()` can be a useful diagnostic aid:

```
      delete a->ptr;
      a->ptr = (void *) (intptr_t) -1;
```

We are not aware of current implementations that do this automatically, but they might exist.

More general lifetime-end pointerzap behaviour, making copies of pointers to lifetime-ended objects `nullptr` across the C runtime, seems unlikely to be practical in conventional implementations. On the other hand, it is arguably desirable for debugging tools that detect erroneous use of pointers after object-lifetime-end to be permitted to do so as early as possible, at the first operation on such a pointer instead of when it is used for an access.

Enable Optimization

Another possible motivation for lifetime-end pointer zap is to enable optimization, e.g. of computations on pointers in cases where the compiler can see they are pointers to lifetime-ended objects.   It seems unlikely to us that this is a significant motivation.

Permit implementation above hardware that traps on loads of pointers to lifetime-ended objects

Modern commodity computer systems do not trap on loads of pointers to lifetime-ended objects, but some historic implementations may have: Intel 80286 for uses of “far pointers” in protected mode, Intel’s iAPX 432, the CDC Cyber 180 (though this is not apparent from extant documentation), and, according to Jones [The New C Standard, p467] the 68000. If past implementations have, then there might be reasons for future implementations to do likewise, though this is rather speculative and should be balanced against the present problem of widespread code idioms that rely on the converse.

In contrast, there has been hardware that enables trapping on dereferencing of invalid pointers, one example being the SPARC ADI feature and another being the ARMv8 MTE feature (slides). Please note that these features do not trap on load, store, and other manipulation of the pointer values themselves. Furthermore, the value representations of the pointers themselves can depend on which allocation produced them, so that two pointers returned from two different calls to `malloc()` might compare not equal even if the corresponding memory addresses are identical.

Specifically, these features use the upper few bits of the pointer values to indicate a memory “color”. If the color of a given pointer does not match that of the corresponding cacheline, any attempted dereferencing of that pointer will trap. This allows `malloc()` and `free()` to change the color of all affected cachlines, so that invalid pointers will (with high probability) trap when dereferenced. Furthermore, the memory colors can be used in such a way as to cause any invalid pointer to memory that has not yet been reused to deterministically trap when dereferenced (the price being a slightly lower probability of trapping when the memory has been reallocated.) As will be shown below, these hardware features are compatible with all concurrent algorithms that we are aware of.

In addition, if two pointers have the same address, but one is invalid and the other is not, one can quite reasonably argue that implementations that cause them to compare not equal are sanctified by existing hardware.

However, these existing hardware have the property that if an invalid and a valid pointer compare equal, it is safe to dereference the invalid pointer. This property is critically important to the correct functioning of the algorithms reviewed in the following section.

Algorithms Relying on Invalid Pointers

This section describes an algorithm that relies on loading, storing, casting, comparing, and (in special cases) dereferencing invalid pointers. (Note that no one is advocating allowing *dereferencing* of invalid pointers unless and until there is a live object at the same address as the lifetime-ended object.) This algorithm dates back to at least 1973, and appears in commonly used code. It would therefore be good to obtain a solution that allows decent optimization and diagnostics while still avoiding invalidating such long-standing and difficult-to-locate algorithms. Additional algorithms relying on use of invalida pointers may be found in WG14 N2369.

It is also worth noting that Kostya Serebryany reports that the Google sanitizer tools do not warn on loads, stores, casts, and comparisons of pointers to lifetime-ended objects because the number of false positives from doing so would be excessive. In other words, code commonly does do *some* computation on such pointers, even if only to print them for debugging or logging.

Categories of Concurrent Algorithms

Although C and C++ do an excellent job of supporting two classes of concurrent algorithms, the fact that pointers to lifetime-ended objects are invalid prevents C and C++ programs that comply with the standard from implementing a third important class of such algorithms. These three classes are listed below, starting with the two that are supported and ending with the as-yet unsupported class:

1. Algorithms that ask permission before both freeing objects and using pointers to those objects. Examples of such algorithms include locking as well as some reference-counting use cases.
2. Algorithms that ask permission before freeing objects, but allow unconditional use of any pointer to any reachable object. Examples of such algorithms include RCU as well as other reference-counting use cases.
3. Algorithms that allow unconditional freeing and pointer use, including the LIFO linked-list push algorithm discussed below. (Again, additional algorithms are discussed in WG14 N2369.)

The fourth possible combination, allowing unconditional freeing of objects, but requires permission to use pointers to those objects, does not yet have any known concurrent algorithms. That aside, C++ should support coding of algorithms in all three of the above categories, not just the first two. In order to emphasize this point, the following section presents one algorithm of many from the third category.

LIFO Singly Linked List Push

This section describes a concurrent LIFO singly-linked list with push and pop-all operations. This algorithm dates back to at least 1973, and is used in practice in lockless code. Note that this code (with `list_pop_all()` and without single node `list_pop()`) is ABA tolerant, that is, it does not require protection from the ABA problem. In addition, when using a simple compiler, it does not require protection from dereferencing invalid pointers, at least from an assembly-language perspective. Please note that this is not the only algorithm with these properties, but is instead a particularly small and simple example of such an algorithm.

```
template<typename T>
class LifoPush {

      class Node {
      public:
            T val;
            Node *next;
            Node(T v) : val(v) { }
      };

      std::atomic<Node *> top{nullptr};

public:
      bool list_empty()
      {
            return top.load() == nullptr;
      }

      void list_push(T v)
      {
            Node *newnode = new Node(v);

            newnode->next = top.load(); // Maybe dead pointer here and below
            while (!top.compare_exchange_weak(newnode->next, newnode))
                  ;
      }

      template<typename F>
      void list_pop_all(F f)
      {
            Node *p = top.exchange(nullptr); // Cannot be dead pointer

            while (p) {
                  Node *next = p->next; // Maybe dead pointer
                  f(p->val); // Maybe dereference dead pointer
                  delete p;
                  p = next;
            }
      }
};
```

The `list_push()` method uses `compare_exchange_weak()` to atomically enqueue an element at the head of the list, and the `list_pop_all()` method uses `exchange()` to atomically dequeue the entire list. From an assembly-language perspective, both ABA and dead pointers are harmless. To see this, consider the following sequence of events:

1. Thread 1 invokes `list_push()`, and loads the `top` pointer, but has not yet stored it into `newnode->next`.
2. Thread 2 invokes `list_pop_all()`, removing the entire list, processing it, and freeing its contents.
3. Thread 1 stores the now-invalid pointer into `newnode->next`. Stepping out of assembly-language mode for a moment, note that this invokes implementation-defined behavior.
4. Thread 2 invokes `list_push()`, and happens to allocate the memory that was at the beginning of the list when Thread 1 loaded the `top` pointer. Although Thread 1's pointer remains invalid from a C++ viewpoint, from an assembly-language viewpoint, its representation once again references a perfectly valid Node object.
5. Thread 1 continues, and its `compare_exchange_weak()` completes successfully because the pointers compare equal. Once again stepping out of assembly-language mode for a moment, note that this invokes implementation-defined behavior.
6. Note that the list is in perfectly good shape: Thread 1's node references Thread 2's node and all is well, again at least from an assembly-language perspective.

7. Continuing the example, Thread 2 once again invokes `list_pop_all()`, removing the entire list. Processing the list is uneventful from an assembly-language perspective, but at the C++ level dereferencing `p->next` invokes undefined behavior due to the fact that Thread 1 stored an invalid pointer at this location.

Note that the `list_pop_all()` member function's load from `p->next` is not a data race. There is no concurrency reason for this load to be in any way special. Although use of std::launder in `list_pop_all()`'s load from `p->next` would address part of the C++-level issue, it would not prevent the implementation-defined behavior that can be invoked when `list_push()` stores a momentarily invalid pointer to this location, nor can it prevent the implementation-defined behavior that can be invoked when `compare_exchange_weak()` accesses this same location.

The following sequence of events shows how memory-coloring hardware would play into this, again from the perspective of assembly language or a simple compiler:

1. Thread 1 invokes `list_push()`, and loads the red-colored `top` pointer, but has not yet stored it into `newnode->next`.
2. Thread 2 invokes `list_pop_all()`, removing the entire list, processing it, and freeing its contents. The color of the memory referenced by Thread 1's `top` pointer changes to orange.
3. Thread 1 stores the now-invalid pointer into `newnode->next`. This stores the pointer, obsolete red color and all, without complaint.
4. Thread 2 invokes `list_push()`, and happens to allocate the memory that was at the beginning of the list when Thread 1 loaded the `top` pointer, so that the color of this memory changes again, this time to yellow.
5. Thread 1 continues, and its `compare_exchange_weak()` fails because the color difference (red versus yellow) is represented by the upper bits of the pointer.
6. However, the `compare_exchange_weak()` loads the new value of the pointer into `newnode->next`, hence updating the color from red to yellow.
7. The next pass through the loop retries the `compare_exchange_weak()`, which now succeeds with the required color match.

Of course, the memory could be freed and reallocated multiple times, resulting in a spurious color match:

1. Thread 1 invokes `list_push()`, and loads the red-colored `top` pointer, but has not yet stored it into `newnode->next`.
2. Thread 2 invokes `list_pop_all()`, removing the entire list, processing it, and freeing its contents. The color of
3. the memory referenced by Thread 1's `top` pointer changes to orange.
4. Thread 1 stores the now-invalid pointer into `newnode->next`. This stores the pointer, obsolete red color and all, without complaint.
5. Other threads repeatedly allocate and free the memory that was originally referenced by the `top` pointer, updating its color, which eventually becomes violet.
6. Thread 2 invokes `list_push()`, and happens to allocate this same memory, updating its color back to red. Thread 1's pointer therefore is once again a perfectly valid pointer from an assembly-language viewpoint.
7. Thread 1 continues, and its `compare_exchange_weak()` completes successfully because the pointers compare equal, colors and all.
8. Note that the list is in perfectly good shape: Thread 1's node references Thread 2's node and all is well, again at least from an assembly-language perspective.
9. Continuing the example, Thread 2 once again invokes `list_pop_all()`, removing the entire list. Because the colors match, processing the list is uneventful from an assembly-language perspective.

This algorithm is thus compatible with actual pointer-checking hardware.

Note that this algorithm requires invalid pointers that happen to point to valid object of an appropriate type be dereferenceable, just as if those pointers were valid. This pointer-zap change does not address this issue, which is to instead be addressed in the provenance work.

Lifetime-end Pointer Zap and Happens-before

If it might be undefined behaviour to load or do arithmetic on a pointer value after the lifetime-end of its pointed-to object, then, in the context of the C/C++11 concurrency model, that must be stated in terms of happens-before relationships, not the instantaneous invalidation of pointer values of the current standard text. In turn, this means that all operations on pointer values must participate in the concurrency model, not just loads and stores.

Lifetime-end Pointer Zap and Representation-byte Accesses

The current standard text says that pointer values become invalid after the lifetime-end of their pointed-to objects, but it leaves unknown the status of their representation bytes (e.g. if read via `char*` pointers). One could imagine that these are left unchanged, or that they also become invalid.

Possible Resolutions

Status Quo

This is of course the “resolution” that results from leaving the standard be. This would leave unstated the ordering relationship between the end of an object’s lifetime and the zapping of all pointers to it. This will also result in practitioners continuing to apply their defacto resolutions.

In fact a number of large pre-C11 concurrent code bases, including older versions of the Linux kernel and prominent user-space applications, avoid these issues for pointers to heap-allocated objects by carefully refusing to tell the compiler which functions do memory allocation or deallocation. At the current time, this prevents the compiler from applying any lifetime-end pointer zap optimizations, but also prevents the compiler from carrying out any optimizations or issuing any diagnostics based on lifetime-end pointer analysis. Of course, this approach may need adjustment as whole-program optimizations become more common, with the GCC link-time optimization (LTO) capability being but one such whole-program optimization. It would therefore be wise to consider longer-term solutions, which is the topic of the next sections.

Eliminate Lifetime-End Pointer Zap Altogether

At the opposite extreme, given that ignoring lifetime-end pointer zap is common practice among sequential C developers, another resolution is to reflect that status quo in the standard by completely eliminating lifetime-end pointer zap altogether. This would of course also eliminate the corresponding diagnostics and optimizations. It is therefore worth looking into more nuanced changes, a task taken up by the following sections.

Limit Lifetime-End Pointer Zap Based on Storage Duration

The concurrent use cases for pointers to lifetime-ended objects seem to involve only allocated storage-duration objects, while the current compiler `nullptr`'ing of pointers at lifetime end appears to apply only to automatic storage-duration objects. A simple and easy to explain solution would therefore be to limit lifetime-end zap to the latter (perhaps also thread-local storage). The biggest advantage of this approach is that it accommodates all known concurrent use cases and also many of the single-threaded use cases. There is some concern that it might limit future compiler diagnostics or optimizations. There is of course a similar level of concern about lifetime-end pointer zap invalidating other algorithms that are not known to those of us associated with the committee.

One can also imagine doing this selectively: introducing some annotation (perhaps an attribute) to identify regions of code that should or should not be subject to lifetime-end pointer zap semantics for allocated storage-duration objects (and/or for all objects).

Note that older versions of the Linux kernel avoid many (but by no means all!) of these issues by the simple expedient of refusing to inform the compiler that things like `kmalloc()`, `kfree()`, `slab_alloc()`, and `slab_free()` are in fact involved in memory allocation.

Limit Lifetime-End Pointer Zap Based on Marking of Pointer Fetches

It was suggested that pointers loaded using C++11 atomics or inline assembly be exempted from lifetime-end pointer zap, and further investigation into existing code prompted volatile loads and stores to be added to this list. This approach would accommodate all verified concurrent use cases, but there is some concern over lock-based algorithms involving pointer revalidation (because the pointers are accessed with locks held, they might well be accessed using plain C-language loads and stores). It also requires adding language to define information flow to the standard, to identify all such pointer instances; this would be complex and require many decisions (analogous to provenance-via-integer semantics).

It was further suggested adding a new marking (perhaps an attribute), which works well for new code, but does not help with existing code.

With or without the new marking, this approach should have minimal effect on compiler optimizations and diagnostics. However, functions to which pointers are passed cannot tell whether those pointers were initially loaded via a marked access. Such functions would need to assume that all pointer arguments were in fact initially loaded via a marked access.

Limit Lifetime-End Pointer Zap to Pointers Crossing Function Boundaries

Martin Uecker suggested that developers should be free to load, store, [cast,] and compare Invalid pointers within the confines of a function (inline or otherwise), but that touching Invalid pointers that have crossed a function-call boundary should be subject to lifetime-end zap. This proposal could be combined with the other proposals that limit lifetime-end pointer zap.

Zap Only Those Pointers Passed to `delete` and Similar

This approach invalidates only those pointers actually passed to deallocators, for example, in `delete p`. In this example, the pointer `p` become invalid, but other copies of that pointer are unaffected, even those within the same function.

Avoid Lifetime-End Pointer Zap by Converting All Pointers to Integers

Although some implementations will (perhaps incorrectly) track pointers through integers, there is a belief that lifetime-end pointer zap should apply only to pointers in pointer form, but not to integers created from pointers. This of course defeats type checking, but such integers could be enclosed in structs with conversion functions, thus providing type checking of a sort.

Although this option might be attractive from the viewpoint of minimizing change to the standard, it has the disadvantage of imposing cognitive load on developers writing some of the most difficult code. Worse yet, it is necessary to convert the integers back to pointers before dereferencing them, which means that use of pointers does not necessarily eliminate lifetime-end pointer zap in many cases. Instead, it merely narrows the window where such zapping can occur, which does not lead to the reliable concurrent software required for today's ubiquitous multicore systems.

Informal Evaluation of Possible Resolutions

A presentation to SC22 WG21 SG12 (C++ Undefined Behavior and Vulnerabilities) resulted in a straw poll favoring elimination of lifetime-end pointer zap altogether.

A presentation at CPPCON 2019 included an informal poll that resulted in 28 votes to eliminate lifetime-end pointer zap altogether, three votes to limit lifetime-end pointer zap to allocated storage-duration objects, and two votes to limit lifetime-end pointer zap based on C11 atomics, inline assembly, and volatile loads/stores. None of the other resolutions received any votes.

This apparent bias in favor of eliminating lifetime-end pointer zap may have been due to the simplicity of the solution, and a possible lack of concern for the effects on compiler diagnostics and optimization, though we note Peter Sewell obtained the same reaction from SG12 from private communication.

After the presentation, Scott Schurr in private communication pointed out that from the C++ Standard N4830 section 6.6.5 Storage duration [basic.stc] paragraph 4.

> 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 (6.7.2). Indirection through an invalid pointer value and passing an invalid pointer value to a deallocation function have undefined behavior. Any other use of an invalid pointer value has implementation-defined behavior.

He feels that the problem we described in the talk, pointers becoming zombies after their storage has gone, is no longer part of the standard. Using such a pointer, other than for indirection or deallocation, is implementation-defined (which is much safer than undefined behavior).

We feel this is covered in our section on What Does the C++ Standard Say which basically it is implementation-defined but there can still be issues. In fact, a LIFO push will dereference a zombie pointer, and it is not just about consistency of comparison cases. In addition, implementation-defined comparisons might well produce random values, which would be inconsistent with most of the algorithms affected by lifetime-end pointer zap.

@@@ Note that some concurrent algorithms require invalid pointers that happen to point to valid object of an appropriate type be dereferenceable, just as if those pointers were valid. This change does not address this issue, which is to be addressed in the provenance work.

Proposed Wording (Based on N4800)

Change *6.6.5 Storage duration [basic.stc]*, paragraph 4 as follows, clearly indicating that as soon as an object's lifetime ends, all pointers to it instantaneously become invalid, and only the dereferencing of those pointers has undefined behavior:

> *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 (6.7.2). Indirection through an invalid pointer value and passing an invalid pointer value to a deallocation function have undefined behavior. Any other use of an invalid pointer value has implementation-defined behaviorthe same effect as would a valid pointer. [34]*
>
> *[34] Some implementations might define that copying an invalid pointer value causes a system-generated runtime fault.*

Change *6.6.5.4.3 Safely-derived pointers [basic.stc.dynamic.safety]*, paragraph 4 as follows, thus reiterating that invalid pointers remain invalid, but invalid pointers still behave the same as valid pointers with the exception of dereferencing:

> *An implementation may have relaxed pointer safety, in which case the validity of a pointer value does not depend on whether it is a safely-derived pointer value. Alternatively, an implementation may have strict pointer safety, in which case a pointer value referring to an object with dynamic storage duration that is not a safely-derived pointer value is an invalid pointer value unless the referenced complete object has previously been declared reachable (19.10.5). [Note: The effect of usingdereferencing an invalid pointer value (including passing it to a deallocation function) is undefined, see 6.6.5. This is true even if the unsafely-derived pointer value might compare equal to some safely-derived pointer value. — end note] It is implementation-defined whether an implementation has relaxed or strict pointer safety.*