ISO/IEC JTC1/SC22/WG21 P1726R3, 2020-02-21

Pointer lifetime-end zap and provenance, too

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

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.

Some of these algorithms also have pointer-provenance issues, however, these issues will be addressed separately.

History

D1726R3 also captures additional discussion during the Prague meeting

- Refine “Enable Optimizations” section even further based on yet more discussions with Richard Smith.
- Add “Zap Pointers Only From the Viewpoint of the EA That Ended the Lifetime” section based on discussions with David Goldblatt.
- Add “debugging” to the pluses and minuses of the various possible resolutions.
- Add more detail to the sections discussing hardware debugging techniques.
- Add verbiage noting the need to pass linked data structures obtained from concurrent algorithms to standard library functions.
- Add additional use cases from WG14 N2443.

D1726R3 captures additional pre-Prague discussion

- Add “hide allocators” section to “possible resolutions” section.
- Add pluses and minuses of the various possible resolutions.
- Refine “Enable Optimizations” section based on discussions with Richard Smith.

Next step: Present to EWG, hopefully at the 2020 Prague meeting.

- SG1 has vetted this paper from a concurrency viewpoint, and supports eliminating lifetime-end pointer zap**.**
- Does EWG have non-concurrency concerns about eliminating lifetime-end pointer zap, and if so, how can these best be addressed? Non-concurrency issues raised and addressed in the past include: (1) Interaction with special-purpose hardware, for example, that detects use-after-free bugs, (2) Interaction with coding errors such as returning the address of an object out of its scope, and (3) Interaction with non-concurrency use cases including tracking pointers to recently freed objects for debugging purposes, debug printing of pointers, pointers as keys for mapping data structures, and checking the value of newly freed pointers as a loop-termination condition.
- Filled out section on optimizers leveraging pointer zap based on January 2020 discussions on the -parallel email reflector.

P1726R1 was presented at Belfast in 2019.

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

- SG1 indicated that eliminating lifetime-end pointer zap is fine as far as concurrency is concerned, and recommended that this paper be sent to EWG.

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.

The WG14 C-Language counterparts to this paper, N2369 and N2443, have been presented at the 2019 London and Ithaca meetings, respectively. These two papers provide much more detailed use cases.

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 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.

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.

Richard Smith noted that optimizers rely on lifetime-end pointer zap in order to safely carry out optimizations where the value of one pointer is used in place of another pointer that has compared equal to it. For example (from Richard's email of January 8th), some current implementations will transform this function:

```
int f(int *p, int *q) {
  // ...
  if (p == q) return *q;
  // ...
}
```

Into this:

```
int f(int *p, int *q) {
  // ...
  if (p == q) return *p;
  // ...
```

```
}
```

This transformation is demonstrated in GCC and LLVM by https://godbolt.org/z/8wvsVT. (Note however that the compiler is prohibited from introducing unspecified or undefined behavior, so the compiler would be prohibiting from introducing this comparison unless both p and q were subsequently used. However, in this case, the comparison is already present, so the compiler is free to assume that both pointers are valid.)

This function might be invoked as follows:

```
int *p = new int; delete p;
int *q = new int(42);
f(p, q);
```

In this case, the lifetime-end pointer zap justifies the optimization, with the value of p having been zapped to that of q.

This function could also be invoked as follows (adapted from Jens Maurer's email of January 9th):

```
struct A {
  int x[1];
  int y;
} a;

int *p = &a.x[1];
int *q = &a.y;
int result = f(p, q);
```

Assuming f() is inlined, p and q will compare not-equal, thus preventing the substitution of pointers in this case, and thus preventing a subsequent dereference of result from invoking undefined behavior. More discussion is required relating to the case where f() resides in a different translation unit than do its callers.

In a subsequent email, Richard showed an example (https://godbolt.org/z/j9c8By) in which removing lifetime-end pointer zap would cause current GCC optimizations to miscompile the program:

```
int f(int *p, int *q) {
  if (p == z)
    return *p;
  if (p == q)
    return *q;
  return 1;
}

int *get() {
  int n = 2;
  return &n;
}
```

```
int h(int *p) {
  int a = 3;
  return f(p, &a);
}

int main() {
  int *p = get();
  return h(p);
}
```

Inlining `f()` into `h()` results in the following:

```
int h(int *p) {
  int a = 3;
  if (p == z || p == &a)
    return *p;
  return 1;
}
```

GCC's dead-store elimination would remove the initialization of `a` in `h()` because it would (reasonably) conclude that the pointer `p` cannot contain the address of `a`. But if the stackframes of `get()` and `h()` are laid out such that `n` and `a` share the same address, the bits contained in `p` really can be the address of `a`, resulting in access to either uninitialized storage or to out-of-lifetime storage.

To Richard's point, one of the proposals calls for lifetime-end pointer zap to remain in place for automatic storage-duration objects. This proposal was not looked upon favorably by people producing models, which is why it has received relatively little emphasis in this paper. And to their point, in theory any example involving automatic storage-duration objects can be straightforwardly transformed into a heap-based example:

1. Add a level of indirection to each automatic storage-duration object.
2. Augment each such object's declaration with a `new`-based initialization.
3. Add a delete for each such object at the point that it goes out of scope.

It remains to be seen how far this theoretical transformation extends into implementations.

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 or might not compare not equal when the corresponding memory addresses are identical. @@@ Need to check with the MTE guys. @@@

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.) @@@ Check the following. @@@ 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 the standard allows implementations to cause them to compare not equal.

Some implementations have a special "all pass" tag value that allows the dereference to proceed regardless of the color of the underlying cacheline. Furthermore, some compilers are expected to "decay" pointer tags to this all-pass value in some circumstances. However, compilers must take care because unconditionally decaying such a tag in a CAS loop can cause the CAS to unconditionally fail. Given the resulting hangs, compilers that carelessly decay tags must be considered to be bug-ridden. @@@ Need to check with the MTE guys. @@@

However, these existing hardware implementations have the property that if an invalid and a valid pointer compare bitwise 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. The following sections describe the following algorithms and classes of algorithms:

- LIFO Linked List push
- Optimized Sharded Locks
- Hazard pointer try_protect
- Checking realloc() return Value
- Identity-only pointers
- Weak pointers in Android

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 and its replacement, N2443.)

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

		// Some users will want to pass list p to standard library functions
		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. (Note that the representation-bits definition of this function's comparison requires the failure.) However, some implementations can "decay" the color bits to the special "all pass" value. If both pointers have been decayed in this way, things play out as described in the last few steps of the next example.
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.

Optimized Hashed Arrays of Locks

This approach uses the time-honored hashed array of locks, but removes the need to acquire locks for statically allocated objects in some cases. For the shallow data structures favored by those writing performance-critical code, this optimization could potentially reduce the number of lock acquisitions by a factor of two, hence is quite attractive.

Holding a particular lock in the array grants ownership of any object whose address hashes to that lock and ownership of any pointer residing in shared memory that references that object, but only if there is at least one pointer residing in memory that references the given object. Dereferencing a given pointer requires hashing that pointer's value, acquiring the corresponding lock, then checking that the pointer has that same value. If the pointer's value differs, the lock must be released and the dereference operation must be restarted from the beginning.

To avoid insertion-side contention, ownership of a `NULL` pointer is mediated by the lock for the structure containing the `NULL` pointer (which might well be just the `NULL` pointer itself). To prevent misordering between the insertion and a concurrent lookup, either: (1) Both the lock on the `NULL` pointer and the to-be-inserted object must be held across the insertion, or (2) Explicit ordering must be provided between the pointer store/load and accesses to the referenced structure. This example code takes the first approach, holding both locks.

Of course, for lookups and deletions, the pointer being dereferenced must be subject to some sort of existence guarantee, for but a few examples:

1. The pointer might be a static global variable whose lifetime is that of the program.
2. The pointer might emanate from an object whose lock is already held.
3. Some other mechanism, such as reference counting, hazard pointers, or RCU might guarantee the pointer's existence. (This sort of use of hazard pointers and RCU in this context was rare back at the time hashed arrays of locks were heavily used.)

For simplicity of exposition, let's assume option 1. For further simplicity, let's choose an extremely simple hash-table structure where each bucket contains a pointer that references either nothing (value of `NULL`) or a single object (non-`NULL` value).

Given a hash function `hash_lock()`, acquiring and releasing a lock for a single structure is straightforward, as shown by the following pseudocode:

```
void acquire_lock(void *p)
{
        int i = hash_lock(p);

        assert(!pthread_mutex_lock(&shard_lock[i]));
}

void release_lock(void *p)
{
        int i = hash_lock(p);

        assert(!pthread_mutex_unlock(&shard_lock[i]));
}
```

If two locks are acquired, deadlock avoidance requires that they be acquired in some order. It is also possible that two distinct structures will hash to the same lock, resulting in slightly more complex lock acquisition and release functions, demonstrated by the following pseudocode:

```
void acquire_lock_pair(void *p1, void *p2)
{
        int i1 = hash_lock(p1);
        int i2 = hash_lock(p2);

        if (i1 < i2) {
                assert(!pthread_mutex_lock(&shard_lock[i1]));
                assert(!pthread_mutex_lock(&shard_lock[i2]));
        } else if (i2 < i1) {
                assert(!pthread_mutex_lock(&shard_lock[i2]));
                assert(!pthread_mutex_lock(&shard_lock[i1]));
        } else {
                assert(!pthread_mutex_lock(&shard_lock[i1]));
        }
}

void release_lock_pair(void *p1, void *p2)
{
        int i1 = hash_lock(p1);
        int i2 = hash_lock(p2);

        if (i1 != i2) {
                assert(!pthread_mutex_unlock(&shard_lock[i1]));
                assert(!pthread_mutex_unlock(&shard_lock[i2]));
        } else {
```

```
                assert(!pthread_mutex_unlock(&shard_lock[i1]));
        }
}
```

Software maintainability considerations clearly prohibit open-coding of these four functions.

To see how lifetime-end pointer zap enters into the picture, consider a simple in-memory part database consisting of a pair of hash tables for part identifiers and names, `idhash` and `namehash`, respectively. To keep things trivial, both identifiers and names are simple integers. Keeping with the tradition of identifiers being assigned by Engineering and names by Marketing, a part might have an identifier but not yet a name. Such a part will be a member of `idhash` but not of `namehash`. A fanciful structure defining such a part might be as follows:

```
struct part {
        int name;
        int id;
        int data;
};
```

Deleting a part must of course remove it from any hash table it is a member of. Deletion by identifier is straightforward, witness the following pseudocode:

```
struct part *delete_by_id(int id)
{
        int idhash = parthash(id);
        int namehash;
        struct part *partp = READ_ONCE(idtab[idhash]);

        if (!partp)
                return NULL;
        acquire_lock(partp);  // Part partp could be deleted and reinserted up to here.
        if (READ_ONCE(idtab[idhash]) == partp && partp->id == id) {
                namehash = parthash(partp->name);
                if (nametab[namehash] == partp)
                        WRITE_ONCE(nametab[namehash], NULL);
                WRITE_ONCE(idtab[idhash], NULL);
                release_lock(partp);
        } else {
                release_lock(partp);
                partp = NULL;
        }

        return partp;
}
```

In the Linux kernel, `READ_ONCE()` is defined roughly as follows:

```
#define READ_ONCE(x) (*(volatile typeof(x) *)&(x))
```

This effect could also be obtained using volatile C++ atomics or inline assembly. Similar observations apply to the Linux kernel's `WRITE_ONCE()` macro.

Note that `parthash()` is the hash function for the `idtab` and `nametab` hash tables, as opposed to the `hash_lock()` function for the sharded locking. If the corresponding hash bucket is empty (`partp` is `NULL`), then there is nothing to delete, hence the `NULL` return. Once the lock is acquired, no other concurrent deletion is possible, but it might be that some other thread deleted the part and then inserted some other part that happened to have the same address and an identifier that hashed to the same bucket. In this case, the `partp` pointer would have been zapped despite still being valid from the viewpoint of a simple compiler. Although this issue might be sidestepped by placing any given data structure in the library, it is necessary for the C language to allow users to construct special-purpose data structures.

Once the lock is acquired, the address and identifiers are checked (using a possibly zapped pointer), and if they match the part is removed from both `nametab` and `idtab` and the lock is released. Otherwise, if the check fails, the lock is released and `partp` `NULL`ed. Either way, the `partp` pointer is returned to the caller, passing back a now-private reference to the part on the one hand or a `NULL`-pointer deletion-failure indication on the other.

Deletion by name is quite similar, but with the roles of idtab and nametab interchanged. The resulting `delete_by_name()` function may be found in github.

Although the check is trivial in this case, it is easy to imagine cases where a more complex check is relegated to a separate function, and furthermore cases where that separate function is supplied by the user.

Note that even read-only access to the value referenced by gp requires locking, as shown in the `lookup_by_id()` and its `lookup_by_bucket()` helper function whose pseudocode is shown below:

```
int lookup_by_id(int id, struct part *partp)
{
        int ret = lookup_by_bucket(idtab, &idtab[parthash(id)], partp);

        if (partp->id == id)
                return ret;
        return 0;
}

int lookup_by_bucket(struct part **tab, struct part **bkt,
                     struct part *partp_out)
{
        int hash = bkt - &tab[0];
        struct part *partp = READ_ONCE(tab[hash]);
        int ret = 0;

        if (!partp)
                return 0;
```

```
        acquire_lock(partp);  // Part partp could be deleted and reinserted up to here.
        if (partp == tab[hash]) {
                *partp_out = *partp;
                ret = 1;
        }
        release_lock(partp);
        return ret;
}
```

These functions operate in a manner similar to the deletion functions, but return a copy of the part rather than deleting the part.

Finally, insertion operates similarly, but as noted earlier must acquire the lock of the bucket pointer and of the to-be-inserted object. Because this trivial example allows only one object per hash bucket, insertion fails when faced with a non-`NULL` bucket pointer, as illustrated by the following pseudocode:

```
int insert_part_by_id(struct part *partp)
{
        return insert_part_by_bucket(&idtab[parthash(partp->id)], partp);
}

int insert_part_by_bucket(struct part **bkt, struct part *partp)
{
        int ret = 0;

        acquire_lock_pair(bkt, partp);
        if (!*bkt) {
                WRITE_ONCE(*bkt, partp);
                ret = 1;
        }
        release_lock_pair(bkt, partp);
        return ret;
}
```

In this case, pointer zap is not an issue because the object referenced by `partp` has not yet been inserted, and therefore cannot be deleted and reinserted.

As with the FIFO push algorithm, hashed arrays of locks are compatible with actual pointer-checking hardware.

More complex linked structures require more sophisticated lock acquisition strategies, which are outlined in the following sections.

How to Handle Lock Collisions?

One approach is to maintain an array of locks already held, along with a count of held locks. This array and count are then passed into acquire_lock(), which checks whether the required lock is already held, and acquires the lock only if it is not already held. Then release_lock() is also passed this array and count, and releases all locks that were acquired.

How to Avoid Deadlock and Livelock?

One approach (heard from Doug Lea) is to use spin_trylock() instead of spin_lock(). If any spin_trylock() fails, all locks acquired up to that point are released, and lock acquisition restarts from the beginning. If too many consecutive failures occur, a global lock is acquired. The thread holding that global lock is permitted to use unconditional lock acquisition, that is, spin_lock() instead of spin_trylock().

Deadlock is avoided because:

1. At most one thread is doing unconditional lock acquisition.
2. Any thread doing conditional lock acquisition will either acquire all needed locks on the one hand, or encounter acquisition failure on the other. In both cases, this thread will release all locks that it acquired, thus allowing the thread doing unconditional acquisition to proceed, thus avoiding deadlock.
3. Any thread that has suffered too many acquisition failures will acquire the global lock and eventually become the thread doing unconditional lock acquisitions, thus avoiding livelock.

Disadvantages

Optimized sharded locks appear to have been used quite heavily in the 1990s, and still see some use. Reasons that they aren't used universally include:

1. Pure readers must nevertheless contend for locks, degrading performance, and, in cases involving "hot spots", also degrading scalability.
2. The hash function will typically result in poor locality of reference, which limits update-side performance.
3. Poor locality typically also results in poor performance on NUMA systems.
4. Much better results are usually obtained through use of a combination of hazard pointers or RCU with a lock residing within each object, as this provides excellent locality of reference and also avoids acquiring locks on any but the data items directly involved in the intended update.

Likelihood of Use

Optimized sharded locks were rederived by Paul based on hearsay from the early 1990s. The likelihood of their use was confirmed by the fact that a randomly selected WG21 member was not only able to derive correct rules for their use based on a vague verbal description, but also able to do so within a few minutes. Given that this person is not a concurrency expert, we assert that someone as intelligent and motivated as that person (which admittedly rules out the vast majority of the population, but by no means all of it) could successfully formulate and use this optimized sharded locking technique. Especially given that this someone would not be under anywhere near the time pressure that this person was subjected to.

It is therefore unnecessary to conduct a software archaeology expedition to find this technique: Given that it has up to a

2-to-1 performance advantage over simple sharded locking, the probability of its use is very close to 1.0. In addition, the code bases in which it is most likely to be used are not publicly available.

Hazard Pointer `try_protect`

Typical reference-counting implementations suffer from performance and scalability limitations stemming from the need for reference-counted readers to concurrently update a shared counter, which results in memory contention, in turn resulting in the aforementioned performance and scalability limitations. This situation motivated the invention of hazard pointers, which can be thought of as a scalable implementation of reference counting. Hazard pointers achieve this scalability by maintaining “inside-out” counters: Instead of a highly contended integer, hazard-pointer readers instead store a pointer to the object to be read into a local *hazard pointer*. The number of such hazard pointers to a given object is the value of that object’s reference count. Because hazard-pointer readers are storing these pointers locally instead of mutating shared objects, memory contention is avoided, thus resulting in good performance and excellent scalability.

However, a given object might be deleted just as a reader is attempting to access it. This means that an attempt to acquire a hazard pointer can fail, just as can happen with many reference-counting schemes. But this also means that hazard-pointers readers need the ability to safely process (but not dereference!) pointers to lifetime-ended objects. Sample “textbook” code for hazard-pointer readers is shown below. This consists of a library part (which could reasonably use special types and markings), followed by a user part, which must be allowed to make use of normal C-language type checking.

The library code is as follows:

```
// Hazard pointer library code
bool hazptr_try_protect_internal(
    hazard_pointer* hp, // Pointer to a hazard pointer
    void** ptr, // Pointer to a local (maybe invalid) pointer
    void* const _Atomic* src) { // Pointer to an atomic pointer
  uintptr_t p1 = (uintptr_t)(*ptr);
  hazptr_reset(hp, p1); // Write p1 to *hp
  /*** Full fence ***/
  *ptr = atomic_load_explicit(src, memory_order_acquire); // Might return invalid pointer
  uintptr_t p2 = (uintptr_t)(*ptr);
  if (p1 != p2) {
    hazptr_reset(hp); // Clear the hazard pointer
    return false; // Caller must not use *ptr
  }
  return true; // Safe for caller to dereference *ptr
}

#define hazptr_try_protect(hp, ptr, src) \
  hazptr_try_protect_internal((hp), (void **)(ptr), (void * const _Atomic *)(src))
```

The approach is to load from the local variable referenced by `*ptr` while converting to `uintptr_t`, storing the result into the hazard pointer, doing a full fence, reloading the pointer, and comparing it to the original. If either the initial caller's load or the final load result in an indeterminate pointer, other portions of the algorithm guarantee that the (naive expectations of the) bit patterns of `p1` and `p2` will differ.

Note that the pointer referenced by ptr in the `hazptr_try_protect()` macro might be indeterminate at the time of the cast.

The following is the user code, written as pseudocode. We don't want to make this code unsafe or error-prone. Note that `user_t` is protectable by hazard pointers.

```
/* Important for the following to remain atomic user_t*
   and not have to become atomic uintptr_t. */
user_t* _Atomic src;

void init_user_data(user_t* _atomic* ptr, value_t v) {
  user_t* p = (user_t*) malloc(sizeof(user_t));
  set_value(p, v);
  atomic_store(ptr, p);
}

void read_only_op_on_user_data() {
  hazard_pointer* hp = hazptr_alloc(); // Get a hazard pointer
  while (true) {
    user_t* ptr = atomic_load(&src);
    // ptr may be invalid here
    if (hazptr_try_protect(&hp, &ptr, &src)) {
      // Safe to dereference ptr as long as *hp protects *ptr
      read_only_op(ptr); // Dereferences ptr.
      break;
    }
  }
  hazptr_free(hp); // Free the hazard pointer for reuse
}

void update_user_data(value_t v) {
  user_t* newobj = (user_t*) malloc(sizeof(user_t));
  set_value(newobj, v);
  user_t* oldobj = atomic_exchange(&src, newobj);
  hazptr_retire(oldobj); // Leads to calling `free(oldobj)` exactly once,
                         // either immediately or later.
  }
}
```

Checking `realloc()` Return Value and Other Single-Threaded Use Cases

The `realloc()` C standard library function might or might not return a pointer to a fresh allocation, and software legitimately needs to know the difference. For example:

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

Without the ability to compare a pointer to a lifetime-ended object, the `realloc()` function becomes rather hard to use. One approach is to cast the pointers to `intptr_t` or `uintptr_t` before comparing them, but not all current compilers respect such casts, as demonstrated by the example code on page 67:8 of SC21 WG14 working paper N2311. In addition, casts have the disadvantage of disabling pointer type checking. It would therefore be good to permit pointer load/store and comparison aspects in cases such as this one.

If the allocated region itself contains pointers to within the region, fixing those up after the `realloc()`
Is even more challenging.

One suggestion was to split `realloc()` into a `try_realloc()` that does in-place extension (if possible), and, if that fails, a `malloc()`/`free()` pair. Outgoing pointers could then be used normally during the time between the `malloc()` of the new location and the `free()` of the old one. It was suggested that most users would not need to know or care about the added complexity of this procedure, and further notes that `realloc()` cannot be used for non-trivial data structures in any case.

Similar use cases from the University of Cambridge Cerberus surveys (see question 8 of 15, and also here and summarized in Section 2.16 on page 38 here) involve:

1. Using the pointer to the newly freed object as a key to container data structures, thus enabling further cleanup actions enabled by the `free()`.
2. Debug printing of the pointer (e.g., using “%p”), allowing the free operation to be correlated with the allocation and use of the newly freed object. Note that it is possible to use things like thread IDs to disambiguate between the pointer to the newly freed object and a pointer to a different newly allocated object that happens to occupy the same memory.
3. Debugging code that caches pointers to recently freed objects (which are thus indeterminate) in order to detect double `free()`s.
4. Some garbage collectors need to load, store, and compare possibly indeterminate pointers as part of their mark/sweep pass.
5. If a pair of pointers might alias, the simplest code would free one, check to see whether the pointers are equal, and if not, free the other.
6. A loop freeing the elements of a linked list might check the just-freed pointer against `NULL` as the loop termination condition. (The referenced blog post suggests use of a `break` statement to avoid such comparisons.)

In short, it is not just obscure concurrent algorithms having difficulty with this “unusual aspect of C". That said, debugging use cases should not necessarily drive the standard and that garbage-collection use cases will usually have at least some implementation-specific code. On the other hand, a feature that purports to improve diagnostics that also causes `printf()` to emit inaccurate and/or misleading results will understandably be viewed with extreme suspicion by a great many C-language developers.

Identity-Only Pointers

This was encountered in the context of SGI’s Open64 compiler many years ago. Hans wishes that he could say that he altered the details to protect somebody or other, but in fact, he just doesn’t remember all the details correctly. So some of this is approximated, preserving the high-level issue.

The compiler was space constrained, since it attempted to do a lot of optimization at link time. As is common for a number of compilers, used region allocation for objects of similar lifetimes, deleting entire regions when the contained data was no longer relevant. At some point it decided that say, a symbol table describing identifier attributes was no longer needed. So the symbol table was deallocated in its entirety.

The rest of the program representation referred to identifiers by pointing into this symbol table. The only information required after the deallocation of the symbol table was to determine whether two identifier references referred to the same identifier. This could still be resolved without the symbol table, and without retaining the associated memory, by just comparing the pointers. And the compiler did so routinely.

(Hans remembers this approach because it foiled his attempt to convert the region-based memory management, which required significant ongoing engineering effort to squash dangling pointer bugs, to conservative garbage collection. The collector would fail to collect the symbol tables, because they were actually still reachable through pointers, just not accessed. Without collecting those, space overhead was excessive.)

Weak Pointers in Android

This is really a C++ example. Correct implementation relies on C++ std::less, which orders arbitrary addresses.

Android provides a reference-count-based weak pointer implementation (https://android.googlesource.com/platform/system/core/+/master/libutils/include/utils/RefBase.h). One of the intended uses of such weak pointers is specifically as a key in a map data structure. They can be safely compared even after all strong pointers to the referent disappear and the referent is deallocated. A weak pointer to a deallocated object at address A will compare unequal to a subsequently allocated object that also happens to occupy address A. Hence a map indexed by such weak pointers can be used to associate additional data with particular objects in memory, without risk of associating data for deallocated objects with new objects.

Comparison of such weak pointers treats the object address as the primary key, and the address of a separate object used for maintaining weak reference information as a secondary key. The second object is not reused while any weak or strong pointers to the primary object remain. The use of the primary key allows ordering to be consistent with std::less ordering on raw pointers. The (primary key) object pointer stored inside a weak pointer is routinely used in comparisons after the referenced object is deallocated. Depending on the particular map data structure that's used and

context, the outcome of comparing a pointer to deallocated memory may or may not matter. But it is currently critical that it not result in undefined behavior.

Since some applications rely on more than equality comparison, so that they can be used in tree maps, I think it is also important that pointers to dead objects can still be compared via std::less (C++) or converted to uintptr_t (C).

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

This section lists a number of potential resolutions, including pluses and minuses of each. The following aspects of each potential resolution are considered:

- Can zap-susceptible concurrent algorithms be reasonably expressed? This must include not only existing algorithms but also yet-to-be-discovered algorithms.
- Can existing code implementing these algorithms be preserved? Such preservation of course need not be reflected in the standard. Instead, the chosen resolution must be amenable to building legacy source code, for example, with a command-line switch that avoids unfortunate optimizations.
- Is modularity preserved? In other words, does the potential resolution in question allow use of the usual C++ abstraction facilities, including function call, templates, separate compilation, and so on? In addition, can data extracted from the various atomic data structures be processed by sequential library functions? (For example, could the list returned by the atomic exchange in `list_pop_all()` be passed to normal list-processing library functions?
- Are compilers still able to use traditional pointer optimizations? Will compilers be able to use yet-to-be-discovered optimizations? Note well that concurrency is also an optimization, so this aspect cannot be considered to have ultimate priority over the other aspects.
- Are compilers and tools still able to do their traditional pointer debugging techniques? Will compilers be able to use yet-to-be-discovered pointer debugging techniques? Note that there are numerous external tools and libraries that do pointer debugging, so this aspect cannot be considered to have ultimate priority over the other aspects.
- Must current compilers be modified?
- Must the standard be changed?

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.

Plusses:

- All optimizations relying on lifetime-end pointer zap still apply.
- All debugging techniques relying on lifetime-end pointer zap still apply.
- No change to existing implementations.
- No change required to standard.

Minuses:

- Zap-susceptible algorithms cannot be expressed reasonably.
- Existing code implementing zap-susceptible algorithms might fail due to optimizations relying on lifetime-end pointer zap.

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, as discussed in the “Enable optimizations” section above. It is therefore worth looking into more nuanced changes, a task taken up by the following sections.

Plusses:

- Allows zap-susceptible algorithms to be written reasonably.
- Allows existing code containing zap-susceptible algorithms to run unchanged.
- Zap-susceptible algorithms can freely use function calls and other C++ features promoting modularity.

Minuses:

- Optimizations relying on lifetime-end pointer zap could not be used, even in code that could tolerate them.
- Debugging techniques relying on lifetime-end pointer zap could not be used, even in code that could tolerate them.
- Possibly significant changes to compiler implementations.

- Requires changes to the standard.

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.

Plusses:
- Allows zap-susceptible algorithms to be written reasonably.
- Allows existing code containing concurrent zap-susceptible algorithms to run unchanged.
- Zap-susceptible algorithms can freely use function calls and other C++ features promoting modularity.

Minuses:
- Optimizations relying on lifetime-end pointer zap could not be used, even in code that could tolerate them.
- Debugging techniques relying on lifetime-end pointer zap could not be used, even in code that could tolerate them. @@@ Needs validation. @@@
- Possibly significant changes to compiler implementations.
- A number of single-threaded use cases remain outside of the standard. (See SC22 WG14 N2443 for a list.)
- Requires changes to the standard.

Zap Pointers Only From the Viewpoint of the EA That Ended the Lifetime

[Per David Goldblatt, as told to and understood by Paul E. McKenney. All errors the property of the latter.]

This approach is intended to codify existing implementations' de-facto handling of pointer zap for concurrent code.

It defines the concept of *conditionally valid*. When an object's lifetime ends, all pointers to that object become conditionally invalid, but only from the viewpoint of the execution agent (EA) that ended that object's lifetime. On all other EAs, whenever a conditionally valid pointer is found to be equal to a valid pointer, the conditionally valid pointer becomes valid, and takes on all the properties of the valid pointer. In addition, there are two potential extensions to this approach.

First, specially marked pointers would remain presumed valid even from the viewpoint of the EA that ended the lifetime of the referenced storage. This situation is expected to prove to be very similar to the marking of pointers and fetches called out below.

Second, the `realloc()` function is a special case in that the pointer passed to it is treated as if it was specially marked.

Plusses:
- Allows zap-susceptible algorithms to be written reasonably.
- Allows most existing code containing zap-susceptible algorithms to run unchanged.
- Zap-susceptible algorithms can freely use function calls and other C++ features promoting modularity.
- All optimizations relying on lifetime-end pointer zap still apply. @@@ Needs validation. @@@
- Debugging techniques relying on lifetime-end pointer zap could still be used. @@@ Needs validation. @@@
- No change to existing implementations. @@@ Needs validation. @@@

- All optimizations relying on lifetime-end pointer zap still apply.

Minuses:
- Requires changes to the standard.

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.

Additionally, any problematic scenario based on automatic storage-duration objects can be converted into a scenario based on allocated storage-duration objects by replacing the declaration of the object with a pointer that is initialized via allocation, and freeing that object when the pointer just before the pointer goes out of scope.

Nevertheless, it is worth reiterating that we do not know of any concurrent algorithms that are inconvenienced by with lifetime-end pointer zapping of automatic storage-duration objects. This property of these algorithms might well help lead to a solution to this problem.

Plusses:
- Allows zap-susceptible algorithms to be written reasonably.
- Allows all known existing code containing zap-susceptible algorithms to run unchanged because zap-susceptible algorithms use heap storage.
- Zap-susceptible algorithms can freely use function calls and other C++ features promoting modularity.

Minuses:
- Some optimizations relying on lifetime-end pointer zap could not be used, even in code that could tolerate them.
- Debugging techniques relying on lifetime-end pointer zap could not be used, even in code that could tolerate them.
- Possibly significant changes to compiler implementations. In fact, it is not clear that this restriction to allocated storage-duration objects is easier on current implementations due to the conversion from automatic to allocated storage duration noted above.
- Requires changes to the standard.

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

This approach exempts loads using C++ atomics (and, outside the standard, via inline assembly or volatile operations) from lifetime-end pointer zap, but only when the destination pointer is marked as exempt from zapping. (Perhaps this should also be extended to provenance.) Note that this section subsumes the "Limit Lifetime-End Pointer Zap Based on Marking of Pointer Fetches" from earlier revisions of this document.

Plusses:

- Allows zap-susceptible algorithms to be written in a not too-unnatural manner.
- Allows current optimizations to be applied freely to unmarked pointers loaded via unmarked fetches.
- Debugging techniques relying on lifetime-end pointer zap could still be used.
- Implementations that provide this functionality can easily provide a command-line argument that causes the compiler to behave as if all pointers and fetches had been so marked, thus preserving existing code.
- Zap-susceptible algorithms can freely use function calls and other C++ features promoting modularity.

Minuses:

- External functions cannot tell which pointers are subject to zapping unless function parameters and return values are also marked.
- Requires changes to the standard, but limited in scope to code requesting the new behavior.

Hide allocations from compiler/optimizer

It is common practice for large code bases using aggressive concurrency to have their own custom allocators and also to hide those allocators from the compiler, so that from the compiler's viewpoint the returned pointer is an unknown pointer, possibly also from an unknown function. This clearly only helps for heap-allocated objects, but this is the primary concern of concurrent algorithms that are inconvenienced by pointer zap.

This proposal is incomplete. More work is needed to determine how much of the problem it actually solves and, if it does help significantly, what changes in wording would be needed to officially bless it.

Plusses:

- If this worked, it would have allowed zap-susceptible concurrent algorithms to be written straightforwardly.
- Relatively small changes are required to existing code containing zap-susceptible concurrent algorithms, especially for such software artifacts already hiding allocation from the compiler.
- Zap-susceptible algorithms can freely use function calls and other C++ features promoting modularity.
- This approach might be useful for benchmarking the effects of optimizations relying on lifetime-end pointer zap.

Minuses:

- Zap-susceptible algorithms would still have problems with (for example) dereferences from one pointer cached in registers and used in lieu of dereferencing another pointer referencing the same memory address. This means that something is required in addition to hiding allocations and deallocations from the compiler, for example, the use of special accesses for dereferences. As of this writing, the only special accesses that are known to be helpful are volatile loads and stores and those in certain types of inline assembly.
- Requires changes to the standard.

- This option reflects existing practice, but the proposal needs more refinement.

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 general. 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.

Plusses:

- All optimizations relying on lifetime-end pointer zap still apply.
- Debugging techniques relying on lifetime-end pointer zap could still be used.
- Zap-susceptible algorithms can freely use function calls and other C++ features promoting modularity, except that library functions expecting pointers as arguments and return values may not be safely used. @@@ Needs validation @@@

Minuses:

- Zap-susceptible algorithms cannot be expressed reasonably.
- Existing code implementing zap-susceptible algorithms might fail due to optimizations relying on lifetime-end pointer zap.
- Race conditions remain where a pointer is zapped just before being converted to an integer or just after being converted from an integer, so this is not a general solution.
- The C++ standard does not yet provide the needed support for converting pointers to integers and back to avoid lifetime-end pointer zap.
- Some implementations must still change given that some have been observed tracking pointer provenance across a conversion from pointer to integer and back (but perhaps this has already been fixed).

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.

Plusses:

- Allows zap-susceptible algorithms whose methods are confined to a single function to be written reasonably.
- Allows existing code containing zap-susceptible algorithms whose methods are confined to a single function to run unchanged.

Minuses:

- Possibly significant changes to compiler implementations.
- Optimizations relying on lifetime-end pointer zap could not be used, even in code that could tolerate them.
- Debugging techniques relying on lifetime-end pointer zap could not be used, even in code that could tolerate them.
- Does not support existing code in which zap-susceptible algorithms span multiple functions, including those that span translation units.
- Requires changes to the standard.

Tabular Summary of Pluses and Minuses

The following table summarizes the pluses and minuses of the proposals presented above.

	Status quo	No zap	Only zap delete	Only zap in EA	Only zap auto	Mark ptr & fetches	Hide alloc & dealloc	Convert ptrs to integers	No zap in func
Express algos?	No	Yes	Yes	Yes	Yes	Yes	Some	Some	Some
Preserve existing code?	No	Yes	Yes	Yes	Yes	no	no	No	Some
Preserve modularity?		Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Preserve opts?	Yes	No	No	Yes?	Some	Yes	yes	Yes	no
Preserve debug?	Yes	No	No?	Yes?	No	Yes	Some	Yes	no
Preserve impls?	Yes	No	No	Yes?	No	no	Yes	yes	no
Preserve std?	Yes	No	No	No	No	no	No	no	No

Informal Evaluation of Possible Resolutions (Historical)

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.