ISO/IEC JTC1/SC22/WG14 N2443, 2019-10-06

# Lifetime-End Pointer 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 indeterminate values 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 collects some of these algorithms and discusses some possible resolutions, ranging from retaining the status quo to completely eliminating lifetime-end pointer zap.

# History

This document is an update of N2369 based on feedback at the 2019 London meeting and on the email reflectors.

# 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, a few of which are presented in the following sections, 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, casted, and compared, due to concurrent removal and freeing of the pointed-to object. 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.

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, and comparison 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 has been in the standard since 1989, and one of the algorithms called out below was put forward in 1973. But its practical consequences will tend to become more severe as compilers do more optimisation, especially link-time optimisation.

# What Does the C Standard Say?

This section refers to Working Draft N2310.

6.2.4p2 states that the value of a pointer becomes indeterminate when the object it references reaches the end of its lifetime:

> The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address [33], and retains its last-stored value throughout its lifetime [34]. If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.
>
> [33] The term "constant address" means that two pointers to the object constructed at possibly different times will compare equal. The address can be different during two different executions of the same program.
>
> [34] In the case of a volatile object, the last store need not be explicit in the program.

3.19.{2,3,4} define “indeterminate value”:

> **indeterminate value**: either an unspecified value or a trap representation
>
> **unspecified value:** valid value of the relevant type where this document imposes no requirements on which value is chosen in any instance. Note 1 to entry: An unspecified value cannot be a trap representation.
>
> **trap representation**: an object representation that need not represent a value of the object type

6.2.6.1p5 states that it is possible that loading a trap representation can result in undefined behavior:

> Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined [51]. Such a representation is called a trap representation.

Thus, after the end of an object lifetime, the C standard no longer requires equality comparison of pointers to that object to be meaningful, as they may be unspecified values. Further, depending on one’s interpretation of the notion of trap representation, which itself may be debatable but which is not the subject of this note, it may be that any load of such a pointer is undefined behavior.

The above appears to be essentially unchanged since C99. In C89/90 (ANSI /ISO 9899-1990), **6.1.2.4 Storage durations of objects** stated something similar, for automatic storage duration objects:

> The value of a pointer that referred to an object with automatic storage duration that is no longer guaranteed to be reserved is indeterminate.

and in **7.10.3 Memory management functions** for allocated storage-duration objects (which is called dynamic storage duration in C++).

K&R (first edition) appears not to say anything analogous.

The separation between object lifetime and storage duration in C++ is more pronounced than in C. Pointers to bytes of allocated storage, compared for equality (or lack thereof) is a supported operation; in particular, from C++17 subclause 6.8 [basic.life], “using the pointer as if the pointer were of type `void*`, is well-defined”, and it is then possible to convert the pointer to a pointer to `unsigned char` and further to compare those pointers for equality.

# 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.

It was noted that some compilers will unconditionally return `NULL` 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, 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.

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

```
      free(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 pointer zap behaviour, making copies of pointers to lifetime-ended objects NULL 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 Indeterminate Pointers

The following sections describe algorithms that rely on loading, storing, casting, and comparing indeterminate pointers. (Note that no one is advocating allowing *dereferencing* of indeterminate pointers.) Many of these algorithms date back decades, and many of them appear 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.

- 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 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.

## LIFO Singly Linked List Push

LIFO singly-linked list with push and pop-all operations.

The push algorithm dates back to at least 1973. Note that this code (with `pop_all` and without single node `pop`) does not require protection from the ABA problem or from dereferencing dangling pointers. The `list_push()` code has been simplified as suggested.

```
typedef char *value_t;

struct node_t {
        value_t val;
        struct node_t *next;
};

void set_value(struct node_t *p, value_t v)
{
        p->val = v;
}

void foo(struct node_t *p)
{
        (*p->val)++;
```

```
}

// LIFO list structure
struct node_t* _Atomic top;

void list_push(value_t v)
{
        struct node_t *newnode = (struct node_t *) malloc(sizeof(*newnode));

        set_value(newnode, v);
        do {
                // newnode->next may have become invalid
        } while (!atomic_compare_exchange_weak(&top, &newnode->next, newnode));
}


void list_pop_all()
{
        struct node_t *p = atomic_exchange(&top, NULL);

        while (p) {
                struct node_t *next = p->next;

                foo(p);
                free(p);
                p = next;
        }
}
```

The `list_push()` method uses `atomic_compare_exchange_weak()` to atomically enqueue an element at the head of the list, and the `list_pop_all()` method uses `atomic_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 `atomic_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 `atomic_compare_exchange_weak()` fails because the color difference (red versus yellow) is represented by the upper bits of the pointer.
6. However, the `atomic_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 `atomic_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 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. Other threads repeatedly allocate and free the memory that was originally referenced by the `top` pointer, updating its color, which eventually becomes violet.
5. 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.
6. Thread 1 continues, and its `atomic_compare_exchange_weak()` completes successfully because the pointers compare equal, colors and all.
7. 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.
8. 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 fully compatible with actual pointer-checking hardware.

## 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:

```
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:

```
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 C-language 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:

```
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) C11 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 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.

```
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. 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. But it is likely that everything here could be done in C with slightly different techniques.

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 replacement of pointer values with indeterminate 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 indeterminate 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 indeterminate.

# 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 `NULL`’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 C11 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

It was suggested that developers should be free to load, store, [cast,] and compare indeterminate pointers within the confines of a function (inline or otherwise), but that touching indeterminate 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. Note that this proposal fails to address cases where pointers subject to lifetime-end zap need to be passed across function boundaries.

## Zap Only Those Pointers Passed to `free()` and Similar

This approach makes indeterminate only those pointers actually passed to deallocators, for example, in `free(p)`. In this example, the pointer `p` becomes 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 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.