ISO/IEC JTC1/SC22/WG21 P1726R5, 2021-07-06

Pointer lifetime-end zap and provenance, too

Authors: Paul E. McKenney, Maged Michael, Jens Maurer, Peter Sewell, Martin Uecker, Hans Boehm, Hubert Tong, Niall Douglas, Thomas Rodgers, Will Deacon, Michael Wong, David Goldblatt, Kostya Serebryany, and Anthony Williams.
Other contributors: Martin Sebor, Florian Weimer, Davis Herring, Rajan Bhakta, Hal Finkel, Lisa Lippincott, Richard Smith, JF Bastien, Chandler Carruth, Evgenii Stepanov, Scott Schurr, Daveed Vandevoorde, Davis Herring, Bronek Kozicki, Jens Gustedt, and Peter Sewell.
Audience: Informational/historical.
Goal: Serve as reference for subsequent focused papers.

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. Similar issues result from certain aspects of C++ *pointer provenance*, for example that stemming from observation of memory allocation. Unfortunately, we do not yet have a precise characterization of the relevant aspects of lifetime-end pointer zap or of pointer provenance, so this paper will use the term *zap* to refer to these to-be-characterized aspects.

This paper presents a number of these algorithms and discusses some possible resolutions, ranging from retaining the status quo to complete elimination of zap.

History

P1726R5 captures discussion and testing since the virtual EWG presentation on July 8, 2020:

- Add a "Terminology" section defining terminology, including defining "zap" as a placeholder for the union of applicable aspects of provenance and lifetime-end pointer zap. Convert the rest of the document to use "zap" in this way.
- Add a statement to the "Zap and Happens-before" section that lifetime-end pointer zap cannot be limited by mundane considerations such as the speed of light if it is to fulfil its role in the C++ language.
- Call out consequent limitations of the "only zap in EA" approach in the "Zap Pointers Only From the Viewpoint of the EA That Ended the Lifetime" section..
- Point out that optimizers might in some cases downgrade dynamic allocation to on-stack variables and call out the optimization and/or marking consequences in the "Mark Allocations and Deallocations" and "Mark Allocations and Deallocations (Zombie Only)" sections.
- Add integer-provenance examples and bugzillas from Peter Dimov, Ville Voutilainen, and Richard Smith to the "Avoid Lifetime-End Pointer Zap by Converting All Pointers to Integers" section.
- Add relevant pointer-comparison to "What Does the C++ Standard Say?" section.
- Add subsections to the "What Does the C++ Standard Say?" section.
- Add a "Sources of Pointer Provenance" section to the discussion of wording from the standard, including references to the work of Peter Sewell's and John Regehr's groups.
- Removed the section containing informal evaluation, moving relevant portions to this section.
- Gathered discussion of the various properties of concurrent applications and placed it in a new section, also adding a diagram from the virtual EWG presentation and furthermore explaining the importance of supporting legacy code.
- Added a "Marking Mechanisms and Syntax" section on different mechanisms and spellings for marking/hiding.
- Updated the code sample in Appendix B.
- Changes since P1726R4 are highlighted.

P1726R4 captures discussion and testing since the post-Prague mailing:

- Based on feedback during the Prague EWG meeting on the topic of pointer-coloring hardware, added results of testing both the LIFO singly linked push algorithm and the optimized sharded locking algorithm on SPARC ADI

hardware and on software emulators for the upcoming ARMv8.5 MTE feature. In all cases, this hardware was compatible with these algorithms.
- Answers questions raised at the Prague EWG meeting, primarily involving pointer coloring.
- Updates a few algorithm/resolution evaluations based on the answers to these questions and on subsequent discussion.
- Add a C++ template using uintptr_t casts to access pointers that are to be shielded from lifetime-end zap. This template is applied to the LIFO push algorithm.
- Add descriptions of the different semantics expected of pointers in different communities.
- Provide a prototype rationale for the prohibition against invented pointer comparisons.

A copy was emailed to the Evolution Working Group email reflector at this point (May 28, 2020). After that, the following changes were made:

- Demoted “Only zap auto” and “No zap in func” to an appendix and removed them from the table. (A copy of the original table resides in this same appendix.)
- Added an option that marks allocations and deallocations and zaps both the zap and provenance except for dereference.
- Added an option that marks selected pointer operations and zaps the zap and provenance except for dereference.
- Added a note stating that the “bag of bits” approach from P2188 is equivalent to “No zap”, where “zap” includes provenance.
- Added a note stating that “No zap” is really “No zap and no provenance”, and renamed to “Bag of bits”.
- Removed the text claiming that valid concurrent algorithms could be subject to racing zaps just before dereferencing the zapped pointer.
- Added a statement that there is no objection to the implementation using integers internally, as long as the user cannot see any non-zap non-provenance difference from the corresponding raw pointer.
- Added more discussions of provenance.
- A straw poll taken in the July 8, 2020 EWG meeting heavily in favor of solving these issues.

P1726R3 captures additional discussion during the Prague meeting
- Refine “Enable Optimizations” section even further based on post-EWG-presentation discussions with Richard Smith.
- Add “Zap Pointers Only From the Viewpoint of the EA That Ended the Lifetime” section based on post-EWG-presentation discussions with David Goldblatt.
- Add “debugging” to the pluses and minuses of the various possible resolutions based on the Prague EWG meeting of potential debugging techniques based on pointer zap.
- Add more detail to the sections discussing hardware debugging techniques in response to discussions at the Prague EWG meeting.
- Add verbiage noting the need to pass linked data structures obtained from concurrent algorithms to standard library functions, based on post-EWG-presentation discussions with Hal Finkel and Chandler Carruth.
- Add additional use cases from WG14 N2443.
- Drop the recommendation and draft wording for zapping the zap based on feedback from the Prague EWG meeting.
- The EWG meeting notes may be found here. A straw poll heavily favored solving these issues.

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 at the 2020 Prague meeting.
- SG1 has vetted this paper from a concurrency viewpoint, and supports eliminating lifetime-end pointer zap**.** **EDIT**: SG1 considers the zapping of pointers referencing lifetime-ended objects to be a bug in the standard.
- 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 C standard since 1989, and the algorithm called out below was put forward in 1973. But its practical consequences will become more severe as compilers do more optimisation, especially link-time optimisation, and especially given the ubiquity of multi-core hardware.

Terminology

- *Bag of bits:* A simple model of a pointer consisting only of its associated address and type, excluding any additional information that might be gleaned from lifetime-end pointer zap and pointer provenance. A simple compiler might well model its pointers as bags of bits.
- *Invalid pointer:* A pointer referencing an object whose storage duration has ended. For more detail, please see the wording in the previous section that excerpted from the standard.
- *Invalid pointer use*: Any use of an invalid pointer (including reading, writing, comparison, casting, and passing to a non-deallocation function) other than passing it to a deallocation function and indirection through it. [Intended to correspond to "*Any other use of an invalid pointer value has implementation-defined behavior.*"]

- *Lifetime-end pointer zap:* Implementations are permitted to act as if the end of an object's storage duration modified all pointers referencing that object.
- *Pointer provenance:* Implementations are permitted to model pointers as more than just a bag of bits.
- *Simple compiler:* A compiler that does no optimization. For the purposes of this paper, results similar to those of a simple compiler can be obtained by marking all memory accesses as `volatile`, albeit potentially at the expense of significant performance degradation.
- *Zap:* The aspects of lifetime-end pointer zap and pointer provenance that are relevant to the categories of algorithms discussed in this paper.
- *Zap-susceptible algorithms:* Any algorithm that would operate correctly when compiled by a simple compiler, but which would need to be coded extremely carefully (for example, by marking all accesses as volatile) to live within the standard.
- *Zombie pointer:* An invalid pointer that happens to contain the same address as a currently valid type-compatible pointer.
- *Zombie pointer dereference*: Indirection through a zombie pointer. [Intended to correspond to the relevant part of "*Indirection through an invalid pointer value and passing an invalid pointer value to a deallocation function have undefined behavior.*"]

What Does the C++ Standard Say?

This section refers to Working Draft N4800.

Lifetime-End Pointer Zap

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. However, there is no provision for converting such an integer back to a pointer if the originally referenced object's lifetime ended, even if there is a new object of compatible type at that same address.

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. Thus, the notion of invalid pointers appears to be a more recent invention.

The earlier notion of pointer identifies a set of memory locations, with types attached to each. The value of the pointer itself may be used regardless of the state of the memory locations, and dereferencing the pointer is permissible as long as the types are sufficiently compatible. Although this notion is quite outdated from the viewpoint of the standard, it is still strongly held in many communities, many members of which prize what they see as its simplicity and utility, in sequential settings, but especially in concurrent settings.

Pointer Comparison

7.6.9 Relational operators [expr.rel], paragraph 4:

> *The result of comparing unequal pointers to objects [82] is defined in terms of a partial order consistent with the following rules:*
>
> *(4.1) - If two pointers point to different elements of the same array, or to subobjects thereof, the pointer to the element with the higher subscript is required to compare greater.*
> *(4.2) - If two pointers point to different non-static data members of the same object, or to subobjects of such members, recursively, the pointer to the later declared member is required to compare greater provided the two members have the same access control (10.8), neither member is a subobject of zero size, and their class is not a union.*
> *(4.3) - Otherwise, neither pointer is required to compare greater than the other.*
>
> *[82] An object that is not an array element is considered to belong to a single-element array for this purpose; see 7.6.2.1. A pointer past the last element of an array x of n elements is considered to be equivalent to a pointer to a hypothetical element x[n] for this purpose; see 6.7.2.*

7.6.9 Relational operators [expr.rel], paragraph 5:

> *If two operands p and q compare equal (7.6.10), p<=q and p>=q both yield true and p<q and p>q both yield false. Otherwise, if a pointer p compares greater than a pointer q, p>=q, p>q, q<=p, and q<p all yield true and p<=q, p<q, q>=p, and q>p all yield false. Otherwise, the result of each of the operators is unspecified.*

7.6.10 Equality operators [expr.eq], paragraph 2:

> *The converted operands shall have arithmetic, enumeration, pointer, or pointer-to-member type, or type std::nullptr_t. The operators == and != both yield true or false, i.e., a result of type bool. In each case below, the operands shall have the same type after the specified conversions have been applied.*

7.6.10 Equality operators [expr.eq], paragraph 3:

> *If at least one of the operands is a pointer, pointer conversions (7.3.11), function pointer conversions (7.3.13), and qualification conversions (7.3.5) are performed on both operands to bring them to their composite pointer type (7.2.2). Comparing pointers is defined as follows:*

(3.1) - If one pointer represents the address of a complete object, and another pointer represents the address one past the last element of a different complete object [83], the result of the comparison is unspecified.
(3.2) - Otherwise, if the pointers are both null, both point to the same function, or both represent the same address (6.7.2), they compare equal.
(3.3) - Otherwise, the pointers compare unequal.

[83] An object that is not an array element is considered to belong to a single-element array for this purpose; see 7.6.2.1.

Although these three paragraphs (and especially 7.6.10p3.2) might seem to guarantee that pointer comparison is based on address, the unfortunate fact is that this paragraph can be superseded by the earlier paragraphs covering invalid pointers. As a result, if the implementation can deduce that at least one of the pointers being compared is invalid, the results of that comparison are implementation defined. In addition, 7.6.10p3.1 allows the implementation to ignore the pointers' addresses when one of them points one past the end of one object and the other is associated with some other object. Interestingly enough, this last allows spurious-seeming equality comparisons in addition to the intended spurious-seeming inequality comparisons, however, this issue does not appear to affect zap-susceptible algorithms.

Sources of Pointer Provenance

In the following situations, the compiler is within its rights to ignore the actual address corresponding to a given pointer's representation bytes:

1. The pointers have been obtained from separate successful non-zero-size allocations.
2. At least one of the pointers references an object whose lifetime (storage duration) has ended. Note that this applies even if that pointer's address now references a new object of compatible type.
3. One of the pointers is a one-past pointer and the other is associated with some other object. There is no known zap-susceptible concurrent algorithm inconvenienced by this, though single-threaded debugging and pointer-tracking use cases might be of concern.
4. The pointers are of incompatible types. There is no known zap-susceptible algorithm inconvenienced by this, and should one appear, casting provides a straightforward resolution. (Some might argue that "bring them to their composite type" will reach `void *`, and thus might further argue that this incompatible-type case never applies.)

This paper thus focuses only on the first two of these possibilities.

Peter Sewell's group has done significant work on the topic of pointer provenance, albeit primarily in the context of the C language. An OOPSLA paper by Juneyoung Lee et al. is also quite pertinent.

In short, the C++ standard's notion of a pointer has rather complex semantics in which, for example, a pair of pointers to objects having identical addresses can nevertheless compare not-equal and vice versa. Given the usual preference for simplicity over complexity, one would expect that there would be some reasons for this semantic complexity, counter-intuitive though these reasons might be to those working with concurrent algorithms. Potential rationales for this complexity are discussed in the following section.

Rationales for Zap Semantics

There are several motivations one might have for 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 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 zap 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.

It is also worth noting that Kostya Serebryany reports that the Google sanitizer diagnostic 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.

Enable Optimization

Another possible motivation for 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 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 proposed 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.

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 cachelines, 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 in the sections discussing the algorithms, 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. And in fact, the closest LLVM comes to decaying tags is the use of stack-offset memory-reference instructions, which ignore tags. Of course, these instructions are used only when the compiler knows that the pointer references the stack, and this technique is therefore not visible to user code, in particular, this technique does not affect pointer comparisons.

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.

Here are some additional references for these hardware features:

- 2019-08-02 Android blog post.
- 2019-08 Arm whitepaper.
- 2019-09 Arm blog post.
- Slides from Intel iSecCon-2018.
- Slides from PLEMM'19.
- Slides presented to RISC-V folks.
- Article in Summer 2019 Usenix ;login.
- CppCon 2018.
- LLVM dev 2018.
- HotSec 2018 (5-minute summary).
- Arxiv paper from Feb 2018.
- Stack instrumentation with ARM Memory Tagging Extension (MTE).

Algorithms Relying on Invalid Pointers

This section describes zap-susceptible algorithms that rely 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 of a compatible type at the same address as the lifetime-ended object.) One of these algorithms 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 section characterizes zap-susceptible algorithms, the section after that gives a rough overview of applications architecture and corresponding constraints, and the final section following that describe specific zap-susceptible algorithms and classes of algorithms.

Categories of Concurrent Algorithms

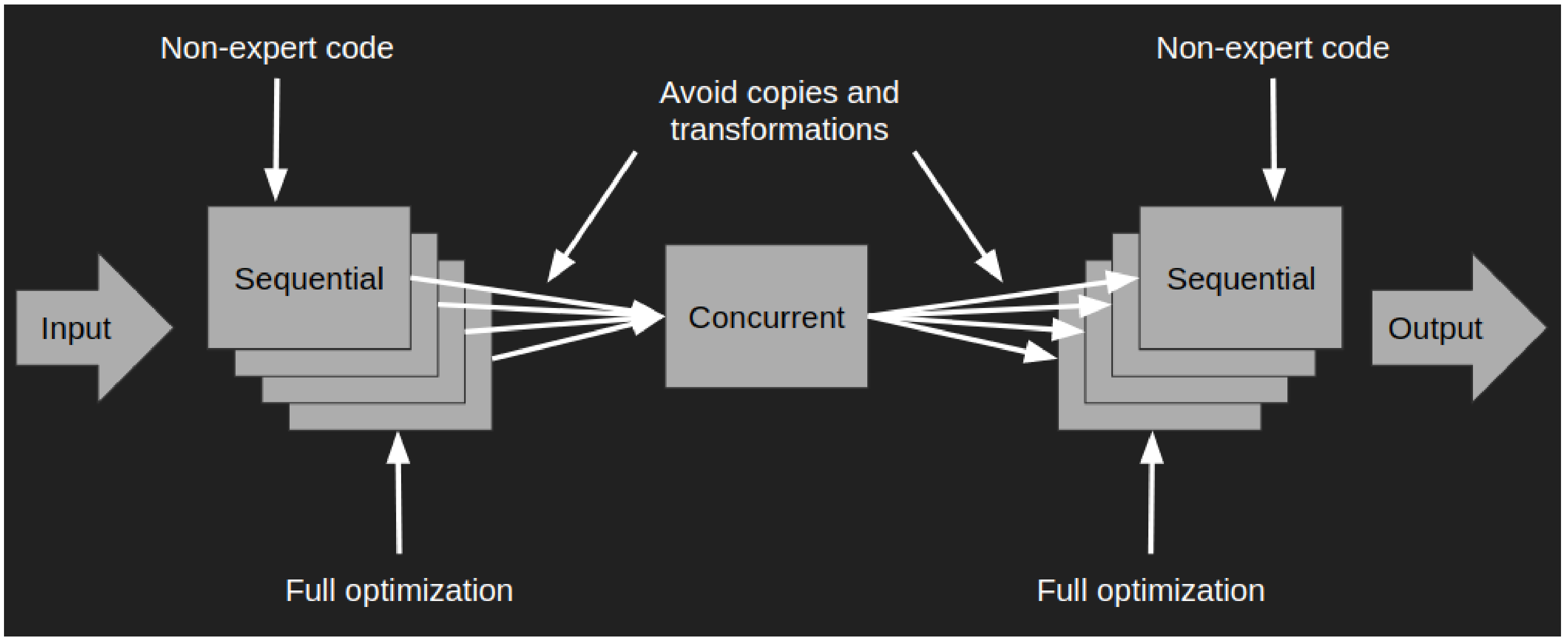
Although C and C++ do an excellent job of supporting two classes of concurrent algorithms, the fact that pointers to lifetime-ended objects are invalid prevents C and C++ programs that comply with the standard from implementing a third important class of such algorithms. These three classes are listed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

Example Concurrent Application Architecture

Before diving into textbook versions of specific zap-susceptible concurrent algorithms, it is worth looking at a typical overall architecture of a large-scale concurrent algorithm, as fancifully depicted in the following figure:



This depicts a large concurrent application that takes in input via a group of threads that do not access shared data, and thus for all intents and purposes comprise sequential code. In many software projects, it is an explicit goal to make the sequential portions represent the bulk of the code, which makes it critically important that these sequential portions can be written by developers with little or (preferably) no concurrency experience or training. It is also important that the compiler be able to aggressively optimize this code.

A smaller portion of the application is fully concurrent, and this portion is the domain of concurrency experts. However, there is frequent communication between the sequential portions and the concurrent portion, and it is critically important that this communication be efficient. In particular, copy and transformation operations must be avoided, even in cases where large linked data structures are passed into the concurrent portion. For example, if a long linked list is passed, it must not be necessary to touch any but the first element of that list, both going into and out of the concurrent portion of the application.

Sadly, it is often necessary to look beyond a given application's technical architecture in order to consider its funding architecture. In particular, applications critical to life as we know it are all too often maintained by organizations that cannot or will not make significant software-development investments. And even organizations having significant software-development resources at their disposal may be hampered by the fact that there is currently no known way of automatically locating zap-susceptible algorithms within large bodies of code. However, 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.

In fact, a number of large pre-C11 concurrent code bases, including various versions of the Linux kernel and prominent user-space applications, avoid zap 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 zap-based optimizations, but also prevents the compiler from carrying out any optimizations or issuing any diagnostics based on lifetime-end pointer analysis for such objects. 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 in fact the topic of this paper.

Zap-Susceptible Algorithms

The following subsections describe these specific 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

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 memory-coloring hardware, which was tested by one of us (Kostya) on SPARC ADI hardware as well as on ARMv8 using HWASAN, which is a software implementation of memory tagging based on the AArch64 top-byte-ignore hardware feature. The tests passed on both systems. The ARMv8 run used an Amazon AWS a1.large instance running Ubuntu, building with clang using the `-fsanitize=hwaddress` compiler command-line argument. The SPARC system used the hardware-specific `-ladimalloc` linker command-line argument. A printf() added to the code on the SPARC system showed the changing color bits within the pointer values, for example printing `0x4fffffff7f01ffb0`, `0x5fffffff7f01ffb0`, `0x6fffffff7f01ffb0`, `0x1fffffff7f01ffb0`, and `0x2fffffff7f01ffb0`, each of which features a different color in the most significant four bits for the same hardware address. This testing therefore found no issues running LIFO singly linked list push on real memory-coloring 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.
Alternative LIFO list code

```
   template <typename T> class LIFOList { // T must have accessible T* next_
     std::atomic<T*> top_{nullptr};
    public:
     void push(T* p) {
1      p->next_ = top_.load(); // Pointer may become invalid before CAS
2      while (!top_.compare_exchange_weak(p->next_, p)) {}
     }
     T* pop_all() { return top_.exchange(nullptr); }
   }; // LIFOList
```

Cases of invalid pointer use in the LIFOList example:
- Line 1 of push: Writing an invalid pointer into p->next_. The pointer became invalid right after being loaded from top_ and before being written to p->next_.
- Line 2 of push: Passing an invalid pointer (p->next_) to compare_exchange
- Line 2 of push, compare_exchange: Reading invalid pointer from p->next_ and comparing with it.
- Line 2 of push, failed compare_exchange: Writing the value found in top_ into p->next_. The value may have just become invalid after being read from top_.

Case of zombie pointer dereference in the LIFOList example:
- If pop_all returns a pointer to the first node in a linked list with more than one node, then nonnull next_ pointers may be zombies.

https://godbolt.org/z/hnod7f

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 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 be invalid despite still perfectly legitimate 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 invalid 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 and was tested by one of us (Kostya) on SPARC ADI hardware as well as on ARMv8 using HWASAN, which again is a software implementation of memory tagging based on the AArch64 top-byte-ignore hardware feature. The tests passed on both systems. The ARMv8 run used an Amazon AWS a1.large instance running Ubuntu, building with clang using the `-fsanitize=hwaddress` compiler command-line argument. The SPARC system used the hardware-specific `-ladimalloc` linker command-line argument. A printf() added to the code on the SPARC system showed the changing color bits within the pointer values, for example printing `0x1fffffff7f2e23a0`, `0x2fffffff7f2e1120`, `0x3fffffff7f2d1120`, `0x4fffffff7f2ce820`, and `0x5fffffff7f2dcfa0`, each of which features a different color in the most significant four bits for the same hardware address. This testing therefore found no issues running optimized hashed arrays of locks on real memory-coloring 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.
  }
}
```

Alternative hazard pointer example:

```
   // User-code
   void reader(std::atomic<T*>& src) {
     T* ptr = src.load(); // ptr may become invalid
     if (hazard_pointer::try_protect(ptr, src))
       f(ptr);
     hazard_pointer::clear();
   }
   void update(std::atomic<T*>& src, T* newobj) {
     T* q = src.exchange(newobj);
     if (q) hazard_pointer::retire(q);
   }
   // Hazard pointer library (simplified)
   std::atomic<T*> _hp;
   void clear() { _hp = nullptr; }
   bool try_protect(T*& ptr, std::atomic<T*>& src) {
     _hp.store(ptr); // ptr value may be or become invalid
     T* p = src.load();
     if (ptr == p) return true;
     ptr = p; return false;
   }
   void retire(T* q) { <Asynchronously call try_reclaim(q) until successful> }
   bool try_reclaim(T* q) {
     if (_hp == q) return false; else { delete q; return true; }
   }
```

Cases of invalid pointer use in the hazard pointer example:
- Line 1 of reader: Writing an invalid pointer into ptr. The pointer may have just become invalid after being loaded.
- Line 2 of reader: Passing an invalid pointer to hazard_pointer::try_protect.
- Line 3 of reader: Passing an invalid pointer to f.
- Line 1 of try_protect: Read an invalid pointer from ptr and store it into hp_
- Line 2 of try_protect: Writing an invalid pointer into p. The value may have just become invalid after being loaded from src.
- Line 3 of try_protect: Reading invalid pointers from ptr and/or p and comparing them. Either or both pointers may be invalid.
- Line 4 of try_protect: Reading invalid pointer from p and writing invalid pointer into ptr.

Cases of zombie pointer dereference in the hazard pointer example:
- Line 3 of reader, call to f(ptr): ptr may be a zombie and f may dereference it.

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.)
7. Passing an integer through an interface that expects a pointer to a callback, for example, by casting the integer to `void *` then casting it back to the original integer type within the callback. Freeing of an object whose address happened to match the integer should not cause the callback to get a different integer than the one that was passed in through the interface. (Yes, template metaprogramming can often avoid the need for this, however, type erasure really is needed from time to time even in C++.)

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).

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.

Unfortunately, the interaction between lifetime-end pointer zap and pointer provenance requires instantaneous action. Therefore, proposals addressing the needs of zap-susceptible algorithms must provide ways of disabling zap.

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. The implementations that we are aware of leave them unchanged.

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, and if so to what extent?
- 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?

A number of these possible resolutions will state that some form of marking is to be used. This marking might be in the source code (for example, use of something like `std::launder`), on the command line, or in the environment (for example, UNIX environment variables), and possibilities are discussed in a later section. Either way, the point of marking is to inform the compiler that it needs to take special care to avoid breaking a zap-susceptible algorithm.

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.

Plusses:
- All optimizations relying on zap still apply.
- All debugging techniques relying on 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 zap-based optimizations.

This option is “Status quo” in the table.

Eliminate Zap Altogether

At the opposite extreme, given that ignoring zap is common practice among sequential C developers, another resolution is to reflect that status quo in the standard by completely eliminating zap altogether, along with pointer provenance. 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 zap could not be used, even in code that could tolerate them.
- Debugging techniques relying on zap could not be used, even in code that could tolerate them. However, this disadvantage must be discounted pending the advent of such techniques.
- Possibly significant changes to compiler implementations.
- Requires changes to the standard.

This option is “Bag of bits” in the table. It is not dissimilar from the approach described in P2188R0.

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` becomes 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, with the exception of those algorithms that use pointer values after `delete`.
- Zap-susceptible algorithms can freely use function calls and other C++ features promoting modularity.

Minuses:

- Optimizations relying on zap could not be used, even in code that could tolerate them.
- Debugging techniques relying on zap could not be used, even in code that could tolerate them. However, this disadvantage must be discounted pending the advent of such techniques.
- 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.

This option is “Only zap delete” in the table.

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

This approach is intended to codify existing implementations' de-facto handling of 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 zap still apply, assuming that this technique can be applied.
- Debugging techniques relying on zap could still be used. However, this advantage must be discounted pending the advent of such techniques.
- No change to existing implementations, assuming that this technique can be applied.
- All optimizations relying on zap still apply.

Minuses:

- Requires changes to the standard.
- No one has yet come up with a specific method of implementing this, although the fact that concurrent code works suggests that it should be possible. However, one complication that must be addressed is that current zap-based optimizations do rely on lifetime-end pointer zap even in cases where the pointed-to object's storage duration was ended by some other EA.

This option is "Only zap in EA" in the table.

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 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 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 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:
- It can be quite difficult to work out what needs to be marked. For example, 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.

One variation on this theme requires marking only for pointers and fetches whose values are later dereferenced, and permits pointers and fetches for values that are only compared (not dereferenced) to remain unmarked.

This option is “Mark ptr & fetches” in the table.

Limit Zap via Marking of Dereferenced Pointer Fetches

This approach implements the marking of pointers and accesses (including writing marked pointers) is to support a template, say, special_ptr<T> (placeholder for a more descriptive name) to mark pointers to type T as exempt from zap and/or provenance analysis. This approach assumes that all pointer operations other than dereference act as would be expected by someone thinking in terms of a simple compiler or an assembler, that is, someone who models pointers as a bag of bits.

If the standard is to confer exemption from zap and/or provenance analysis on intptr values, a possible implementation of special_ptr can be as follows:

```
template <typename T> class special_ptr {
  uintptr_t iptr_;
 public:
  constexpr special_ptr(T* p = nullptr) : iptr_(reinterpret_cast<uintptr_t>(p)) {}
  constexpr T* get() { return reinterpret_cast<T*>(iptr_); }
  static constexpr special_ptr& ref(T*& p) { return reinterpret_cast<special_ptr<T>&>(p); }
};
```

If the standard exempts T* pointers operated on using references to special_ptr<T> from zap and/or provenance analysis, then the following can be a valid implementation of the LIFO list algorithms:

```
template <typename T> class LIFOList { // T must have accessible T* next_
  std::atomic<special_ptr<T>> top_{nullptr};
 public:
  void push(T* p) {
    special_ptr<T>::ref(p->next_) = top_.load(); // Pointer may become invalid before CAS
    while (!top_.compare_exchange_weak(special_ptr<T>::ref(p->next_), special_ptr<T>(p))) {}
  }
  T* pop_all() { return top_.exchange(special_ptr<T>(nullptr)).get(); }
};
```

In the above example the marking of pointers is needed only in expert code implementing the LIFO list algorithm. The extended example in Appendix A demonstrates the distinction between expert and non-expert code. It is important to avoid viral markings that would need to leak into non-expert code. The example in Appendix A demonstrates that allocation and dereference of zombie pointers could happen in non-expert code, whereas expert code is where a valid pointer could become invalid and then zombie.

Plusses:
- Allows almost all zap-susceptible algorithms to be written in a natural manner. Only those algorithms needing to dereference invalid pointers need to mark the affected pointer fetches.
- Allows current pointer-zap optimizations to be applied freely to unmarked pointers loaded via unmarked fetches.
- Debugging techniques relying on 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 pointer fetches had been so marked, thus preserving existing code.
- Zap-susceptible algorithms can freely use function calls and other C++ features promoting modularity.

Minuses:
- Some implementations might need changes to provenance-based pointer-value optimizations.
- Requires changes to the standard, but limited in scope to code requesting the new behavior.
- Prevents use of some optimizations, but the value of those optimizations has not been shown.
- In the absence of global marking, it can be quite difficult to work out what needs to be marked. For example, external functions cannot tell which pointers are subject to zapping unless function parameters and return values are also marked.

This option is “Mark ptr & fetches (deref)” in the table.

Mark Allocations and Deallocations

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.

Instead of hiding, this approach marks the allocations and (if necessary) the deallocations as well. The implementation would then avoid making pointer-provenance and lifetime-end pointer-zap assumptions based on marked allocations and deallocations. Another way of looking at this is that the compiler would be forbidden from assuming that newly allocated memory is not aliased by other allocations.

Plusses:
- Allows zap-susceptible concurrent algorithms to be written straightforwardly.
- Relatively small changes are required to existing code containing zap-susceptible concurrent algorithms, and the code that must be updated is easily searched for.
- 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 zap.
- This option follows long-standing existing practice.

Minuses:
- Requires changes to the standard, but these should be relatively confined.
- Requires changes to implementations, but these should be relatively small.
- Implementations must either refrain from carrying out optimizations that move objects from heap allocations to the stack on the one hand, or must mark the resulting stack-allocated objects so as to avoid making zap-based assumptions about any pointers referencing these objects.

Note that the compiler is prohibited from inventing comparisons between pointers. Without this prohibition, some argue that a compiler could transform this code:

```
p = malloc(sizeof(*p));
initialize_me(p);
do_something(p->a, p->b);
free(p);
q = malloc(sizeof(*q)); // p and q have the same address.
initialize_me_differently(q);
do_something(q->a, q->b)
```

Into something like this:

```
p = malloc(sizeof(*p));
initialize_me(p);
do_something(p->a, p->b); // p->a and p->b are cached in registers.
free(p);
q = malloc(sizeof(*q)); // p and q have the same address.
initialize_me_differently(q);
if (p == q)
      do_something(p->a, p->b);  // Old values from registers!!!
```

This transformation would have the effect of incorrectly propagating the zapping of an invalid pointer to a properly used live pointer, which would break conforming code. However, many implementers argue that the compiler is permitted to take full advantage of any user-supplied pointer comparisons, thus resulting in the first of the minuses. [This example has not yet been vetted by implementers, so please take it with a quantity of salt.]

This option is “Mark alloc & dealloc” in the table.

Mark Allocations and Deallocations (Zombie Only)

This proposal requires algorithms that dereference invalid pointers to mark allocations (and perhaps also deallocations) of the pointed-to objects. These markings would allow the compiler to avoid zap-based optimizations on these pointers. This approach assumes that all pointer operations other than dereference act as would be expected by someone thinking in terms of a simple compiler or an assembler, that is, someone who models pointers as a bag of bits.

Plusses:

- Allows zap-susceptible concurrent algorithms to be written straightforwardly, with marking required only for those algorithms that dereference invalid pointers.
- Relatively small changes are required to existing code containing zap-susceptible concurrent algorithms, especially for such software artifacts already hiding allocations and frees from the compiler. Implementations that provide command-line flags causing all allocation and deallocation invocations to be implicitly marked can correctly build existing code bases without changes to the source code.
- 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 zap.

Minuses:

- Requires changes to the standard.
- Requires changes to the implementations.
- Prevents use of some optimizations, but the value of those optimizations has not been shown.
- Implementations must either refrain from carrying out optimizations that move objects from heap allocations to the stack on the one hand, or must mark the resulting stack-allocated objects so as to avoid making zap-based assumptions about any pointers referencing these objects.

This option is “Mark (de)alloc (zombie)” in the table.

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 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 templated classes with conversions, thus restoring type checking in the context of C++. Note that some people consider this templated approach to be a special case of marking pointers and fetches.

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. In addition, the need to move linked data structures back and forth between sequential and concurrent processing means that there must be some sort of zero-cost en-masse conversion of all pointers in those linked data structures.

Plusses:

- All optimizations relying on zap still apply.
- Debugging techniques relying on zap could still be used.

Minuses:

- Zap-susceptible algorithms cannot be expressed reasonably. (However, converting pointers to integers might be a reasonable implementation strategy, as long as the integer nature of these pointers is completely hidden from the user.)
- Existing code implementing zap-susceptible algorithms might fail due to optimizations relying on lifetime-end pointer zap.

- Although zap-susceptible algorithms can use some function calls and other C++ features promoting modularity, library functions expecting pointers as arguments and return values may cannot be used safely. This situation is especially acute for functions whose arguments or return values comprise linked structure containing arbitrarily large numbers of pointers that are subject to 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 zap.
- Some implementations must still change given that some have been observed tracking pointer provenance across a conversion from pointer to integer and back:
    - Peter Dimov's example code.
    - GCC Bugzilla 49330 from 2011 (hat trick to Ville Voutilainen).
    - LLVM Bugzilla 34548 from 2017 (hat trick to Richard Smith).
    - LLVM Bugzilla 46380 from 2020 and a video of Juneyoung Lee's corresponding presentation to the 2019 EuroLLVM Developers' Meeting (hat trick to Richard Smith).

This option is "Convert ptrs to integers" in the table.

Tabular Summary of Pluses and Minuses

The following table summarizes the pluses and minuses of the proposals presented above in graphical form. Red "No" is a strong negative. Dark yellow "No" is a medium negative. Light yellow "No" is a weak negative. Black "Yes" is a strong positive. Dark yellow "Yes" is a medium positive. Light yellow "Yes" is a weak positive. Question marks ("?") indicate some differences of opinion are likely. Of course, one could reasonably argue that differences of opinion are inevitable across the entire table.

	Status quo	*Bag of bits*	Only zap delete	Only zap in EA	Mark ptr & fetches	Mark ptr fetches (zombie)	Mark alloc & dealloc	Mark (de)alloc (zombie)	Convert ptrs to integers
Express algos?	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Some
Preserve: existing code?	No	Yes	Some	Yes	no *	Yes **	Yes **	Yes **	No
sequential?	No	Yes	Yes	Yes	No	Yes	Yes **	Yes	No
modularity?		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Some
optimizations?	Yes	No?	No?	Yes?	Yes	Yes?	Yes?	Yes?	Yes
debugging?	Yes	?	?	Yes?	Yes	Yes	Yes	Yes	Yes
impls?	Yes	No?	No?	Yes?	no?	no?	no?	no?	yes?

standard?	**Yes**	No	No	No	no	no	No	no	no

* Better code preservation is offered by approaches requiring marking only of to-be-dereferenced pointers/fetches.
** Assuming implementations provide command-line options to mark all applicable instances (see below).

Use of command-line options to affect an entire executable can work extremely well in the following cases:

- Buildable source code is available, and the build system can be modified to add the command-line option.
- For any source code that is not available or for which the build system cannot be reasonably modified, a third party can be influenced to do the needed work.

The first case often holds for standalone executables and for executables built entirely from open-source components. The second case holds when there are third-party proprietary components, but where sufficient influence exists over the corresponding vendors of those third-party proprietary components.

In other cases, the command-line option might not be attractive.

Marking Mechanisms and Syntax

This section is based on discussions at the July 8, 2020 EWG meeting that discussed the P1726R4 version of this working paper. The first subsection describes the underlying library and language mechanisms that might be employed by any marking syntax, and the second subsection surveys syntactic possibilities.

This section proposes the aforementioned `usable_ptr<T>` syntax, and also proposes that atomic variables and operations be immune from pointer invalidation in order to ease transition of legacy code and also to ease production of new code. This section further proposes that successful CAS operations be treated as if they updated the value of the pointed-to expected value when this value is a pointer.

This allows `usable_ptr<T>` to be defined in terms of Clang address spaces, an upgraded `uintptr_t`, or an upgraded `atomic<T>` (but with the default `memory_order` being `memory_order_relaxed`).

Marking Mechanisms

This section looks at the following mechanisms for protecting zap-susceptible algorithms.

- Use of or cast through `volatile`. This has to work for `volatile` to function in its device-driver role, but unless `volatile` is needed for some non-invalid-pointer reason, it produces inefficient code for many use cases.
- Use of or cast through `atomic<T *>`. But unless `atomic<T *>` is needed for some non-invalid-pointer reason, it needlessly restricts how the variables can be used.
- Successful `compare_exchange` operations are modeled as if they updated their “old” value for purposes of provenance and determining pointer invalidity.

- Cast through `intptr_t` or `uinptr_t`. This has been the advertised method of dealing with invalid pointers for a very long time, so needs to be made to work. But use of integer types for pointers defeats type checking, thus inviting a large class of bugs.
- Provide separate "address spaces".

These will be judged based on the following criteria:

- Solves the entire problem?
- If this cannot solve the entire problem, can it be dispensed with?
- Requires changes to implementations?
- Requires changes to the standard and/or the various provenance proposals?
- ABI compatibility for linking separately compiled code?
- Overhead? Zero overhead is preferable, with the gold standard of course being negative overhead.
- Suitable for whole-program use? In other words, is a command-line argument practical.
- Useful for objects that are not dynamically allocated? (This is a nice-to-have.)
- What syntactic touch points are required?

These criteria are called out in the plusses, minuses, and syntax sections for each proposal.

This judgment will be based on GCC 10.0 and Clang 10.0.0 builds of the following code sample, which was derived from a twitter post by JF Bastien:

```
void call(void*);

void hmm(size_t sz) {
  void* p = malloc(sz);

  func(p);
  void* q = malloc(sz);
  if (p == q)
    call(p);
}
```

Current implementations can and do elide the `call(p)` as shown by https://godbolt.org/z/64fW4x.

Use of or cast through `volatile`

The much-despised but ever-helpful `volatile` keyword is not all that well defined by the standard, but consider device drivers that use an in-memory command buffer containing pointers controlling DMA operations. Any compiler able to correctly build such device drivers must forgive and forget any invalidation associated with a pointer value subject to a `volatile` access.

One approach is to add a `volatile` to the definition of the pointer `p`. This restores the `call(p)` as shown by https://godbolt.org/z/68ccGW. Note that GCC's code is two instructions shorter due to the fact that `p` is forced to memory, removing the need for register save/restore operations. Of course, shorter might not be faster, especially given the store-load sequence, and it is easy to construct examples where the overhead of repeated accesses to `p` overwhelms any savings that might accrue from avoiding register save/restore operations.

In theory, it is possible to provide a compiler command-line argument that causes all variables to be treated as if they were `volatile`, but the overhead would almost always be intolerable. All implementations capable of correctly compiling device drivers will handle `volatile` as required to protect zap-susceptible algorithms. The subject of changes to the standard surrounding `volatile` has traditionally resulted in spirited discussions, and there is no reason to believe this time would be different. Then again, what is life without the occasional spirited discussion?

Another alternative is to use a `volatile` cast when accessing pointer `p` for the comparison, for example, by defining a `NO_ZAP()` macro as follows:

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

Applying this macro to the `if` condition's load from `p` restores the `call(p)` as shown by https://godbolt.org/z/vP1aj3, however, it results in an increase in the number of instructions emitted by GCC because this approach incurs the overhead of both register save/restore operations and a flush to memory. Again, it is easy to construct examples where this overhead is larger than (and, for that matter, smaller than) that of the original zap-susceptible code.

Strangely enough, one more alternative is to apply the `NO_ZAP()` macro to the `if` condition's load from `q` (not to be confused with `p`). This approach also restores the `call(p)` as shown by https://godbolt.org/z/8KGxoW.

Pluses:
- Requires no changes to implementations capable of correctly building modern device drivers.
- Requires no changes to the standard, at least for those making the perhaps brave assumption that the standard has a good and sufficient definition of `volatile`. However, it likely requires that the various provenance proposals treat `volatile` pointers and casts the same as they do casting pointers through integers.
- Works for all storage durations, including on-stack variables.
- Provides full ABI compatibility, courtesy of its long use in device drivers.

Minuses:
- Increases overhead for some important use cases, and these will be unacceptable in some fast-path use cases.
- Not suitable for whole-program use. Unless it is OK to eliminate almost all optimizations.
- Therefore, this cannot solve the entire problem. Nevertheless, it must solve much of the problem in order to meet other requirements, it has been used to solve this problem for quite some time, and both the standard and all correct implementations support this. It is therefore an important component of the overall solution.

Syntax: (1) Declare the pointer `volatile`, (2) Use `volatile` casts to load and store the pointer, similar to the Linux kernel's `READ_ONCE()` and `WRITE_ONCE()`, (3) Straightforward library implementation of `usable_ptr<T>`, or (4) The primitives proposed in P1382R1: volatile_load<T> and volatile_store<T> .

Extensions to `std::launder`

This approach is similar to that for `volatile`, but with `NO_ZAP()` defined as follows:

```
    #define NO_ZAP(x) (std::launder(x))
```

Except that neither GCC 10.1 nor clang 10.0.0 is happy with this: https://godbolt.org/z/cWYMGG. GCC (but not clang) is happy with the following alternative, suggested by Richard Smith and Marshall Clow:

```
#define NO_ZAP(x) (__builtin_launder(x))
```

Unfortunately, it was clang that was showing the provenance-based optimizations: https://godbolt.org/z/jqE5eq. The solution to these build errors is to add casts to the `malloc()` expressions as noted by Hubert: https://godbolt.org/z/xqfvE4. However, this does not prevent invalidation, though perhaps some similar `std::launder_nozap` might provide these additional semantics.

Pluses:

- If it worked, there would likely be no additional overhead.
- If it worked, it would likely work for all storage durations, including on-stack variables.

Minuses:

- Requires changes to implementations, as current `std::launder` implementations do not erase the source of the storage from the compiler. It is instead used to permit changes to a given object, such as modifying `const` members.
- Not suitable for whole-program use unless wholesale pointer-fetch substitution is acceptable.
- Requires changes to the standard, given that the current wording does not allow use on invalid pointers. It also requires that the various provenance proposals treat `std::launder` (or similar) of pointers the same way as they do casts of pointers through integers.

Syntax: Casts on sources of the pointer values and on the accesses using the pointers, depending on the zap-susceptible algorithm in question.

If it worked, the `usable_ptr<t>` could be implemented in terms of `std::launder`.

Cast through `intptr_t` or `uinptr_t`

This approach is similar to that for `volatile`, but with `NO_ZAP()` defined (for example) as follows:

```
#define NO_ZAP(x) ((typeof(x) *)(intptr_t)(x))
```

This is subject to the issues called out in the earlier section on casts through integers. In particular, it does not help in the case of clang 10.0.0: https://godbolt.org/z/1axs9e.

This situation might be addressed by the PNVI-ae-udi alternative presented in Peter Sewell's group's work. Alternatively, an OOPSLA paper by Juneyoung Lee et al. suggests that pointers obtained from integers should have wildcard provenance, which is similar to the PNVI alternative presented by Peter Sewell's group. The PNVI-ae-udi alternative requires that the pointers be "exposed" before being converted to integers, for example, by being printed, bytewise accessed, or (presumably) stored via a volatile operation.

Pluses:

- If it worked, there would be no change in overhead.

- If it worked, it would work for all storage durations, including on-stack variables.
- If it worked, it would support ABI compatibility.

Minuses:
- Requires changes to some implementations, for example, clang 10.0.0.
- Requires “exciting” provenance-related changes to the standard, for which there are several proposals.
- It is not clear how to adapt this to whole-program use.
- If the above issues are addressed, this would solve the whole problem. In addition, this has been the advertised solution to this problem for some time, so it only makes sense to take this approach work.

Syntax: If it worked, it would provide a straightforward library implementation of `usable_ptr<t>`.

Hide allocations and/or deallocations

This approach (perhaps paradoxically) adds markings that hide allocations and deallocations from the compiler. One way to do this is (in the specific case of the `malloc()` function) to change the name of `malloc()` to `hidden_malloc()`, as shown here: https://godbolt.orgodboltg/z/seGj56. This succeeds in protecting zap-susceptible algorithms.

Pluses:
- No increase in overhead for concurrent use cases.
- Suitable for whole-program uses with substantial informal implementation experience.

Minuses:
- Possibly increased overhead for programs containing both concurrency and code for which provenance-based optimizations are important.
- Does nothing for objects that are not dynamically allocated.
- Requires changes to implementations, but these changes are believed to be minor.
- Requires changes to the standard, which would need to ignore allocator-based provenance of pointers obtained from marked allocations or whose objects have been subjected to marked deallocations.

Syntax: Numerous possibilities, including command-line argument, cast on allocations, and new set of allocator APIs.

This can be implemented in terms of the `usable_ptr<t>` approach. Implementations are encouraged to add outside-of-standard options to hide allocators in legacy code.

Provide separate allocator pools

This approach is quite similar to hiding allocations and/or deallocations from the compiler, but in addition allows run-time diagnostics to easily determine whether a given block of allocated memory is intended for use by zap-susceptible algorithms. This information might help to improve runtime diagnostics. A name change suffices here as well, at least from the perspective of simple test scenarios: https://godbolt.org/z/P4o7Px. This succeeds in protecting zap-susceptible algorithms.

Pluses:
- No increase in overhead.
- Suitable for whole-program uses.

Minuses:

- Does nothing for objects that are not dynamically allocated.
- Requires changes to implementations, but these changes are believed to be minor.
- Requires changes to the standard. In particular, it requires that the various provenance proposals treat pointers to objects residing in these separate pools in the same way as they do integers obtained from casts of pointers.

Syntax: TBD.

Provide separate “address spaces”

Clang provides a notion of address spaces that are enabled by its `-DDO_ADDR` command-line argument and used via the `__attribute__((address_space()))` construct. Some address space numbers are predefined, for example, 256, 257, and 258 cause a corresponding pointer to be relative to the x86 gs, fs, and ss registers, respectively. Using smaller-numbered address spaces can disable certain aspects of provenance: https://godbolt.org/z/sM7315.

Similar functionality is said to be implemented by other compilers, and page 37 (PDF page 45) of ISO/IEC TR 18037 describes similar address spaces in the context of embedded controllers.

Pluses:

- Little or no increase in overhead.
- Requires no changes to implementations already supporting this notion of address space.
- Works for all storage durations, including on-stack variables?
- It should support ABI compatibility.

Minuses:

- Whole-program use might conflict with other uses of this attribute.
- Requires changes to the standard. In particular, it requires that the various provenance proposals treat pointers to objects residing in these separate address spaces in the same way as they do integers obtained from casts of pointers (assuming the aforementioned “exciting” changes to the standard).
- If the above issues are addressed, this would solve the whole problem.

Syntax: If it worked, it would provide a straightforward library implementation of `usable_ptr<t>`.

Use of or casts through atomic<T *>

Given that zap-susceptible concurrent code uses atomic pointers, it is worth exploring the possibility of making `atomic<T *>` pointers free of invalidation semantics. However, this does not necessarily restore the `call(p)` as shown by https://godbolt.org/z/cP9G3G, which means that changes to implementations would be required. In theory, it is also possible to provide a compiler command-line argument that causes all pointers to `T` to be treated as if they were `atomic<T *>`, but the overhead would almost always be intolerable.

Another tool would be a casting regimen similar to that used for the approach using `intptr_t` or `uintptr_t`:

```
#define NO_ZAP(x) \
({ \
        std::atomic<typeof(x)>___p = x; \
        (typeof(x))___p; \
    })
```

Similar to the `atomic<T *>` pointers, this does not necessarily restore the `call(p)` as shown by https://godbolt.org/z/E8n85d.

Pluses:

- If it worked, it would work for all storage durations, including on-stack variables.
- If it worked, it would provide at least a partial solution for and existing code that implements zap-susceptible algorithms using `atomic<T *>` pointers.
- If it worked, it would support ABI compatibility.

Minuses:

- Requires changes to implementations.
- Requires changes to the standard. For example, it requires that the various provenance proposals treat atomic pointers in the same way as they do integers obtained from casts of pointers.
- Many use cases will see increases in overhead, and these will be unacceptable in some fast-path use cases.
- Not suitable for whole-program use, unless it is OK to increase overhead due to the added memory-fence instructions required by the default `memory_order_seq_cst`.
- Therefore, this cannot solve the entire problem. Nevertheless, it greatly eases implementation of zap-susceptible algorithms, and should therefore be provided.

Syntax: The existing `atomic<T *>` will work in most cases. In addition, it would provide a straightforward library implementation of `usable_ptr<t>`.

Modeled Update of Old Value by `compare_exchange`

This approach is specific to uses of the `compare_exchange` family of atomic read-modify-write operations. The idea is that a successful `compare_exchange` operation notionally updates the old value despite the fact that this value does not change. This would be notionally similar to casting a pointer to this old value to volatile, then using this volatile pointer to load the previous value then store it back, but without actually doing the load or the store. The effect is to forget any provenance or pointer-invalidity information that might have been previously attached to the old value.

Pluses:

- If it worked, there would be no change in overhead.
- If it worked, it would work for all storage durations, including on-stack variables.
- If it worked, it could be adapted to whole-program use, but only for this `compare_exchange` use case.
- If it worked, it would support ABI compatibility.

Minuses:

- Possibly requires changes to existing implementations.
- Requires provenance-related changes to the standard.
- Therefore, this cannot solve the entire problem. Nevertheless, it greatly eases implementation of zap-susceptible algorithms, and should therefore be provided.

Syntax: No change needed.

Marking Syntax

This section only lays out requirements. Specific details will depend on the marking mechanism, among other things. Possibilities discussed thus far include:

- Pointer-wrapper template class.
- Type qualifier similar to const.
- C++ attribute (but note viral issues from transactional-memory experience!).
- Wrapper of some sort for allocations.
- Transform, perhaps similar to `std::launder`.
- Command-line argument (outside of the standard).

Many of the mechanisms described in the previous section can already be expressed using some existing syntax, but there are likely to be advantages to abstracting into some syntax. For example, different platforms might favor different mechanisms. However, please note that all implementations generating code for a given platform should use the same mechanism in order to permit libraries built with a given implementation to be usable by code built with other implementations.

Appendix A: Example using template marking exempt pointers and accesses

```
https://godbolt.org/z/fWbdf3asv

#include <atomic>
#include <iostream>

///
/// LIBRARY CODE
///

// Template to mark pointers that may become invalid
template <typename T> class usable_ptr {
  uintptr_t _iptr;
 public:
  constexpr usable_ptr(T* p = nullptr) : _iptr(reinterpret_cast<uintptr_t>(p)) {}
  constexpr T* get() { return reinterpret_cast<T*>(_iptr); }
  static constexpr usable_ptr& ref(T*& p) { return reinterpret_cast<usable_ptr<T>&>(p); }
}; // usable_ptr

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

///
/// USER CODE
///

// Expert 1: Implements LIFOList using usable_ptr.
/// Comments include code with unmarked pointers
template <typename T> class LIFOList { // T must have accessible T* next_
  /// std::atomic<T*> top_{nullptr};
  std::atomic<usable_ptr<T>> top_{nullptr};
 public:
  void push(T* p) {
    /// p->next_ = top_.load(); // Pointer may become invalid before CAS
    usable_ptr<T>::ref(p->next_) = top_.load(); // Pointer may become invalid before CAS
    /// while (!top_.compare_exchange_weak(p->next_, p)) {}
    while (!top_.compare_exchange_weak(usable_ptr<T>::ref(p->next_), usable_ptr<T>(p))) {}
  }
  /// T* pop_all() { return top_.exchange(nullptr); }
  T* pop_all() { return top_.exchange(usable_ptr<T>(nullptr)).get(); }
}; // LIFOList

// Non-expert 1: Doesn't know about LIFOList or usable_ptr.
```

```
struct Node { Node* next_; };

// Non-expert 2: Knows about Node but doesn't know about LIFOList and usable_ptr.
void consume_list(Node* p) {
  while (p) {
    std::cout << p << " ";
    Node* next = p->next_;  // p may be a zombie pointer (starting from the second iteration).
    delete p;
    p = next;
  }
  std::cout << std::endl;
}

// Non-expert 3: Knows about Node and LIFOList but doesn't know about usable_ptr.

LIFOList<Node> l;

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

// Non-expert 4: Knows only about construct_list and consume_list
int main() {
  consume_list(construct_list());
}
```

Appendix B: Deprecated Options

This appendix lists options that have been discussed, but which have little support.

Limit 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 invalidation 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 pointer invalidation preventing straightforward use of 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 pointer-invalidation 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 invalidation 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 pointer invalidation could not be used, even in code that could tolerate them.
- Debugging techniques relying on pointer invalidation 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.

This option is “Only zap auto” in the table.

Limit 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 invalidation. This proposal could be combined with the other proposals that limit pointer invalidation.

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 pointer invalidation could not be used, even in code that could tolerate them.
- Debugging techniques relying on pointer invalidation 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.

This option is “No zap in func” in the table.

Full Summary Table

The following table summarizes the pluses and minuses of the proposals presented above in graphical form. Red "No" is strong negative. Dark yellow "No" is a medium negative. Light yellow "No" is a weak negative. Black "Yes" is a strong positive. Dark yellow "Yes" is a medium positive. Light yellow "Yes" is a weak positive. Question marks ("?") indicate some differences of opinion are likely. Of course, one could reasonably argue that differences of opinion are inevitable across the entire table.

	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**	Some	**Yes**	**Yes**	no *	no	No	Some
Preserve modularity?		**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	Some	No
Preserve opts?	**Yes**	No?	No?	Yes?	Yes?	**Yes**	yes?	**Yes**	no?
Preserve debug?	**Yes**	?	?	Yes?	No?	**Yes**	Yes?	**Yes**	no?
Preserve impls?	**Yes**	No?	No?	Yes?	No?	no?	**Yes**	yes?	no?
Preserve std?	**Yes**	No	No	No	No	no	No	no	No

* Better code preservation is offered by approaches requiring marking only of to-be-dereferenced pointers/fetches.

Focused Summary Table

The following table evaluates a few selected options and their combination as follows:

- Solves the entire problem? ("Suffices?")
- If this cannot solve the entire problem, can it be dispensed with? ("Optional?")
- Requires changes to implementations? ("Std chg?")
- Requires changes to the standard and/or the various provenance proposals? ("Impl chg?")
- ABI compatibility for linking separately compiled code? ("ABI compat?")
- Overhead? Zero overhead is preferable, with the gold standard of course being negative overhead. ("Zero ovhd?")
- Suitable for whole-program use? In other words, is a command-line argument practical. ("Whole pgm?")
- Useful for objects that are not dynamically allocated? All of the proposals are useful, so there is no column in the table for this option.
- What syntactic touch points are required? ("Syntax?")

	Suffices?	Optional?	Std chg?	Impl chg?	ABI compat?	Zero ovhd?	Whole pgm?	Syntax?
`volatile`	N	N	N	N	Y	n	N	N/A
`atomic<T>`	N	n	Y	?	?	Y	N	N/A
`usable_ptr<T>`	y	N	Y	Y	N/A	Y	y	Library
implicit `usable_ptr<T>`	Y	n	N/A	Y	N	N	Y	N/A
All of the above	Y	nN	Y	Y	?Y	Y	yYN	Library