ISO/IEC JTC1/SC22/WG21 P3790R1, 2025-08-08

Pointer lifetime-end zap proposed solutions

Bag-of-bits pointer class

Authors: Paul E. McKenney (paulmckrcu@kernel.org), Maged Michael (maged.michael@gmail.com), Jens Maurer, Peter Sewell, Hans Boehm, Hubert Tong, Niall Douglas, Thomas Rodgers, Will Deacon, Michael Wong, David Goldblatt, Kostya Serebryany, Anthony Williams, Tom Scogland, JF Bastien, Daniel Krügler, and David Tenty.
Other contributors: Martin Sebor, Florian Weimer, Davis Herring, Rajan Bhakta, Hal Finkel, Lisa Lippincott, Richard Smith, JF Bastien, Davis Herring, Chandler Carruth, Evgenii Stepanov, Scott Schurr, Daveed Vandevoorde, Davis Herring, Bronek Kozicki, Jens Gustedt, Peter Sewell, Andrew Tomazos, Davis Herring, Martin Uecker, and Jason McGuiness.
Audience: SG1, LEWG, EWG.
Goal: Provide a `launder_ptr_bits()` function and a `ptr_bits<T>` template class to enable zap-susceptible algorithms.

Abstract

The C++ standard currently specifies that all pointers to an object become invalid at the end of its lifetime [basic.life]. Although this permits additional diagnostics and optimizations which might be of some value, it is not consistent with long-standing usage, especially for a range of concurrent and sequential algorithms that rely on loads, stores, equality comparisons, and even dereferencing of such pointers. Similar issues result from object-lifetime aspects of C *pointer provenance*.

We propose (1) the addition to the C++ standard library of the function launder_ptr_bits() that takes a pointer argument and returns a prospective pointer value corresponding to its argument; and (2) the addition to the C++ standard library of the class template std::ptr_bits<T> that is a pointer-like type that is still usable after the pointed-to object's lifetime has ended.

Please note that this paper does not propose adding bag-of-bits pointer semantics to the standard. However, in the service of legacy code, it is hoped that implementers provide such semantics, perhaps via some facility such as a command-line option that causes all pointers to be exempt from lifetime-end pointer invalidity.

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 decades 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 and WG21 P1726R5 for the corresponding C++ language details.)

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 this issue's practical consequences will become more severe as compilers do more optimisation, especially link-time optimisation, and especially given the ubiquity of multi-core hardware.

This paper proposes straightforward specific solutions. This paper was split out from P2414R8 ("Pointer lifetime-end zap proposed solutions: atomics and volatile"), which contains proposals for atomic and volatile pointer accesses.

Terminology

- *Bag of bits:* A simple model of a pointer consisting only of its associated address and type, excluding any additional information that might be gleaned from lifetime-end pointer zap and pointer provenance. A simple compiler might well model its pointers as bags of bits. For the purposes of this paper, a non-simple compiler can be induced to treat pointers as bags of bits by marking all pointer accesses and indirections as `volatile`, albeit with possible performance degradation.
- *Invalid pointer:* A pointer referencing an object whose storage duration has ended. For more detail, please see the "What Does the C++ Standard Say?" section of P1726R5, particularly the reference to section 6.7.5.1p4 [basic.stc.general] of the standard ("When the end of the duration of a region of storage is reached, the values of all pointers representing the address of any part of that region of storage become invalid pointer values"). In the C standard, such a pointer is termed an *indeterminate pointer*.
- *Invalid pointer use*: Any use of an invalid pointer (including reading, writing, comparison, casting, passing to a non-deallocation function), and indirection through it. [Intended to correspond to [basic.stc.general] p4 "*Any other use of an invalid pointer value has implementation-defined behavior.*"]
- *Lifetime-end pointer zap:* An event causing a pointer to become invalid, or, in WG14 parlance, indeterminate. Because this is a WG21 document, the term *becomes invalid* is used in preference to "lifetime-end pointer zap", however, text that needs to cover both C++ and C will use the term "lifetime-end pointer zap", "pointer zap", or just "zap".
- *Pointer provenance:* Implementations are permitted to model pointers as more than just a bag of bits.
- *Prospective pointer value:* A pointer value corresponding to an object whose lifetime might not have started, including a pointer to an object whose region of storage has not yet been created. A correct algorithm will not compare or dereference a prospective pointer until after an appropriately typed object's lifetime starts at the address indicated by the pointer's value. Note that comparison of a prospective pointer's value representation is permitted, for example, as carried out by the `.compare_exchange` member function. One way to produce a prospective pointer is to cast a valid pointer to `uintptr_t` and then cast it back to the same pointer type. Implementations that do not provide `uintptr_t` can support these changes via the as-if rule: They need not convert pointers to and from integers, but they must discard any provenance not represented in the representation value as if they supported `uintptr_t`. For more information, please see P2434R4. Note however that P3248R3 ("Require [u]intptr_t") was forwarded to LWG for C++29, so this might become a non-issue.
- *Simple compiler:* A compiler that does no optimization. For the purposes of this paper, results similar to those of a simple compiler can be obtained by treating all pointers as bags of bits.
- *Zap-susceptible algorithm:* An algorithm that relies on invalid pointer use and/or zombie pointer dereference.

- *Zombie pointer:* An invalid pointer whose value representation happens to correspond to the same memory address as a currently valid pointer to an object of compatible type.
- *Zombie pointer dereference*: Indirection through a zombie pointer. [The relevant part of the standard being [basic.stc.general] p4: "*Indirection through an invalid pointer value and passing an invalid pointer value to a deallocation function have undefined behavior.*"]

What We Are Asking For

In order to support a number of critically important algorithms, this paper proposes a `launder_ptr_bits()` function and a `ptr_bits<T>` template class to provide a convenience mechanism for encapsulating the pair of `reinterpret_cast<>` operations used to create a prospective pointer value as described in the "Consequences for pointer zap" section of P2434R4.

Note that this paper does not propose blanket bag-of-bits pointer semantics, despite a great many users being strongly in favor of such semantics (P2188R1). It is therefore hoped that implementers will provide some facility to cause pointers to be treated as bags of bits from a pointer-invalidity viewpoint, perhaps by implicitly treating all pointer types as if they were `ptr_bits<T>`. This would be helpful for legacy code.

The following sections provide more detail on this proposal. Those interested in seeing a wider array of historical options are invited to review P1726R4, P1726R5, and P2188R1.

Possible polls:

1. Do we want a `launder_ptr_bits()` function that provides a convenience mechanism for encapsulating the pair of reinterpret_cast<> operations used to create a prospective pointer value as described in the "Consequences for pointer zap" section of P2434R4?
2. Do we want a `ptr_bits<T>` convenience mechanism for encapsulating the pair of `reinterpret_cast<>` operations used to create a prospective pointer value as described in the "Consequences for pointer zap" section of P2434R4?
3. Do we want `launder_ptr_bits()` usable in constexpr expressions? (Author position: "no" unless and until `reinterpret_cast<>` operations are usable in constexpr expressions.)
4. Do we want expressions involving `ptr_bits<T>` usable in constexpr expressions? (Author position: "no" unless and until `reinterpret_cast<>` operations are usable in constexpr expressions.)

Detailed Proposal

As noted earlier, this paper proposes: (1) A `launder_ptr_bits()` function and (2) A `ptr_bits<T>` template class.

A `launder_ptr_bits()` Function

This section describes a convenience mechanism for encapsulating the pair of `reinterpret_cast<>` operations used to create a prospective pointer value as described in the "Consequences for pointer zap" section of P2434R4.

A launder_ptr_bits() function takes a pointer as its argument and returns the corresponding prospective pointer value. This function can be based on a reinterpret_cast pair as suggested in P2434R4 (see the "Consequences for pointer zap" section), for example, by using an uintptr_t data member private to ptr_bits<T>. As suggested by the "Proposal" section of that same paper, the memcpy() function could instead be used.

The launder_ptr_bits() function cannot be used in constexpr contexts because reinterpret_cast<> cannot be used in constexpr contexts. However, should reinterpret_cast<> become usable in constexpr contexts, then launder_ptr_bits() would also be usable in such contexts.

A ptr_bits<T> Template Class

This section describes a convenience mechanism for encapsulating the pair of reinterpret_cast<> operations used to create a prospective pointer value as described in the "Consequences for pointer zap" section of P2434R4.

A ptr_bits<T> template class may be used to mark pointers in order to forgive pointer invalidity. The provenance discussion gives a solid basis for this, but there is a need to treat normal user-supplied pointers as if they were of the ptr_bits<T> template class. This template class can be based on a reinterpret_cast pair as suggested in P2434R4 (see the "Consequences for pointer zap" section), for example, by using an uintptr_t data member private to ptr_bits<T>. As suggested by the "Proposal" section of that same paper, the memcpy() function could instead be used. However, this proposal allows for use of the reinterpret_cast pair by avoiding marking the ptr_bits<T> template class's constructors as constexpr. However, the ptr_bits<T> template class cannot be used in constexpr contexts only because reinterpret_cast<> cannot be used in constexpr contexts. Therefore, should reinterpret_cast<> become usable in constexpr contexts, then ptr_bits<T> would also become usable in such contexts.

Example

User Tracking of Pointers and realloc()

(Note: It is not clear that this example survives the transition from angelic pointer provenance to non-deterministic pointer provenance. The possibly invalid argumentation is nevertheless retained for future in-depth review.)

Hans's realloc() example compares the return value of realloc() with its argument to determine whether other pointers to the pointed-to object need to be updated. Here is Hans's original code:

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

This code can be simplified as follows:

```
T* p;

q = realloc(p, newsize);
```

```
        if (q != p)
              p = q;
```

And then this simplified code can be fixed using `ptr_bits<T>` as follows:

```
        ptr_bits<T> p;

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

This will re-evaluate provenance on `p` according to its value representation any time that `p` would otherwise be an invalid pointer.

Wording

The following sections describe adding a `ptr_bits<T>` class, comparison operators, a `launder_ptr_bits()` function, and a hash specialization to the `<memory>` header, referring to N4993: C++ Working Draft.

Add `ptr_bits<T>`, comparisons, `launder_ptr_bits()`, and hash specialization

n.m Class ptr_bits **[ptr.bits]**

n.m.1 General **[ptr.bits.general]**

A pointer's value representation can contain address and provenance information, and the C++ implementation might maintain additional provenance information that is not contained in that value representation. This additional provenance information can be problematic for certain types of concurrent algorithms and debugging code, which might need comparison and hashing functions to be consistent with the value representation. This consistency is especially important for concurrent algorithms using the `compare_exchange` family of functions.

This template class relies on conversions from integer to pointer producing pointers (if any) that result in defined behavior, even if all pointers having the corresponding value representation are invalid at that point in time. However, in order to preserve important optimizations, if a conversion from integer to pointer happens before the operation (such as an allocation) that makes such a pointer valid, the implementation is free to produce an invalid pointer.

```
namespace std {
  template <typename T>
  class ptr_bits {
    ptr_bits_internal iptr; // Exposition only
  public:

    // n.m.2, member functions
    ptr_bits(T* ptr) noexcept;
    ptr_bits(nullptr_t);
    T& operator*() const noexcept;
    operator T*() const;
    T* operator->() const;
    T* get() const noexcept;
    T* operator=(T* ptr) noexcept;
  };

  // n.m.3, non-member functions
  template<class T>
    friend const bool operator<=>(const ptr_bits<T>& a, const ptr_bits<U>& b)
noexcept;
    friend const bool operator<=>(const ptr_bits<T>& a, U *b) noexcept;
    friend const bool operator<=>(const ptr_bits<T>& a, nullptr_t b) noexcept;
    T* launder_ptr_bits(T* ptr) noexcept;

  // n.m.4, hash support
  template<class T> struct hash<ptr_bits,T>>;
}
```

`ptr_bits_internal` is an unspecified type that satisfies the requirements for converting a pointer to an integer [expr.reinterpret.cast]/4.

[Note: It is possible that the `ptr_bits_internal` type is not otherwise available to the user, though the common choice is likely to be `uintptr_t`. -- end note]

[Note: Accesses to `iptr` are not observable behavior. -- end note]

n.m.2 Member functions [ptr.bits.members]

`ptr_bits(T* ptr = nullptr) noexcept;`

Effects: Initializes `iptr` with `reinterpret_cast<ptr_bits_internal>(ptr)`.

`ptr_bits(nullptr_t) noexcept;`

Effects: Initializes `iptr` with `0`.

`T& operator*() const;`

Returns: `*get()`.

`operator T*() const noexcept;`

Returns: `get()`.

`T* operator->() const noexcept;`

Returns: `get()`.

`T* get() const noexcept;`

Returns: `reinterpret_cast<T*>(iptr)`.

`T* operator=(T* ptr) noexcept;`

Effects: Assigns `reinterpret_cast<ptr_bits_internal>(ptr)` to `iptr`.

[Note: The result can differ from the original pointer value ([expr.reinterpret.cast]). -- end note]

Returns: `get()` using the new value of `iptr`.

n.m.3 Non-member functions [ptr.bits.func]

`friend const bool operator<=>(const ptr_bits<T>& a, const ptr_bits<U>& b) noexcept;`

Returns: `a.iptr <=> b.iptr`.

`friend const bool operator<=>(const ptr_bits<T>& a, nullptr_t b) noexcept;`

Returns: `a.iptr <=> 0`.

`friend const bool operator<=>(const ptr_bits<T>& a, U *b) noexcept;`

Returns: `a.iptr <=> ptr_bits(b).iptr`.

`template<class T> T* launder_ptr_bits(T* ptr) noexcept;`

Returns: `reinterpret_cast<T*>(reinterpret_cast<ptr_bits_internal>(ptr))`.

[Note: The result can differ from `ptr` ([expr.reinterpret.cast]). -- end note]

n.m.3 Hash support [ptr.bits.hash]

`template<class T> struct hash<ptr_bits,T>>;`

For an object `p` of type `ptr_bits<T>`, `hash<ptr_bits<T>>()(p)` evaluates to the same value as `hash<typename ptr_bits<T>::element_type*>()(p.iptr)`.

[Note: The operation of `hash<ptr_bits<T>>()(p)` should be consistent with `operator<=>()`. -- end note]

Note that, strictly speaking, `p.iptr` is ill-formed because `iptr` is a private member. Is this OK?

History

D3790R1:

- Select `launder_ptr_bits()` and `ptr_bits<T>`.
- More names for the class formerly known as usable_ptr<T>.

P3790R0:

- Split the `launder_bag_of_bits_ptr()` function and the `bag_of_bits_ptr<T>` template class out of P2414 ("Pointer lifetime-end zap proposed solutions: atomics and volatile") in order to permit P2414 to make progress while naming and other details are hammered out.
- Add implementer encouragement to make the hash specialization consistent with the `<=>` operator. By definition, the `<=>` operator is consistent with the value-representation comparison used by the `compare_exchange` functions, as needed by concurrent algorithms.

P2414R8:

- Following Anthony Williams's P2188R1 ("Zap the Zap: Pointers are sometimes just bags of bits"):
  - Rename `usable_ptr<T>` to `bag_of_bits_ptr<T>`.
  - Rename `make_usable_ptr()` to `launder_bag_of_bits_ptr()`.

D2414R8:

- Move the non-direct-pointer access example to D3347R3.
- Update terms based on Davis Herring feedback.
- Add name-selection guide.
- Note relationships to `constexpr` contexts.

P2414R7:

- Rebase discussion onto "P2434R4 Nondeterministic pointer provenance".
- Add a LIFO Push algorithm with exposed pointers to help demonstrate the limits of pointer-zap ergonomics if there is no angelic provenance.

D2414R7:

- Switch comparisons to spaceship operator (`<=>`) based on feedback from SG1 and from Daveed Vandevoorde at the 2025 Hagenberg meeting.
- Additional changes based on feedback from SG1 at the 2025 Hagenberg meeting:
  - Make only `T&` dereference operator be `constexpr`.
  - Make `operator==()` be `const` rather than `constexpr`.
  - Remove class `D` from hash specification.
  - Change name from `make_ptr_prospective()` to `make_usable_ptr()`.
  - Implementations lacking `uintptr_t` can still implement this via the as-if rule.
- Make the `nullptr_t` constructor for `usable_ptr<T>` initialize `iptr` to zero in order to make it compatible with the comparison operators.

P2414R6:

- Apply Frank Birbacher feedback:

    - Add page numbers.
    - Fix unbalanced parentheses and double negative in LIFO Push code sample.
    - Add `operator->()`, `operator=()`, and `get()` to `usable_ptr<T>` synopsis.
    - Add comparison operators to `usable_ptr<T>`.
    - Add `nullptr_t` constructor to `usable_ptr<T>`.
    - Define `make_ptr_prospective()` in terms of `usable_ptr<T>`.
- Apply Mark Hoemmen feedback by removing `constexpr` from `usable_ptr<T>` constructors and adding a sentence explaining why.
- Apply Bryan St. Amour feedback by supplying a `usable_ptr<T>` specialization for `std::hash`.

P2414R5:
- Update references to P2434 to the latest version.
- Move Martin Uecker from author list to contributor list at his request.
- Apply SG1 feedback:
    - Add more alternative names for `usable_ptr<T>`.
    - Fix declaration of * operator for `usable_ptr<T>`.
    - Fix code for casting to `volatile atomic<T>`.
    - Reviewed P2434R1 for "words of power" for prospective provenance, but found none.
- Apply EWG feedback:
    - Rework atomics wording to avoid the need to otherwise duplicate all atomic operations in [atomics.types.pointer].
    - Add similar wording to [atomics.ref.pointer].
    - Rework volatile wording (also in response to private communications with Davis Herring).
    - Extract the pointer-handling material to P3347R0 Invalid/Prospective Pointer Operations.
- Updated from "representation bytes" to "value representation" to track N4993: C++ Working Draft.
- Updated the definition of "prospective pointer value" to cover the possibility that multiple instances of an object might be created and deleted before that pointer's provenance is established.

P2414R4:
- Updated based on the June 24, 2024 St. Louis SG1 review:
    - Fix numerous typos.
    - Drop discussion of defining load, store, and arithmetic operations on invalid and prospective pointers to allow them to be in their own paper.
    - Add function as well as class.
- Added draft wording and updated per Daniel Krügler feedback.
- Move the history section to the end of the paper.

D2414R4:
- Updated based on the June 24, 2024 St. Louis EWG review and forwarding of P2434R1: Nondeterministic pointer provenance from Davis Herring and subsequent discussions:
    - The prospective-pointer semantics remove the need for a provenance fence, but add the need for a definition of "prospective pointer".
    - Leverage prospective pointer values.
    - Adjust example code accordingly.

P2414R3:

- Includes feedback from the March 20, 2024 Tokyo SG1 and EWG meetings, and also from post-meeting email reflector discussions.
- Change from reachability to fence semantic, resulting in provenance_fence().
- Add reference to C++ Working Draft [basic.life].

P2414R2:

- Includes feedback from the September 1, 2021 EWG meeting.
- Includes feedback from the November 2022 Kona meeting and subsequent electronic discussions, especially those with Davis Herring on pointer provenance.
- Includes updates based on inspection of LIFO Push algorithms in the wild, particularly the fact that a LIFO Push library might not have direct access to the stack node's pointer to the next node.
- Drops the options not selected to focus on a specific solution, so that P2414R1 serves as an informational reference for roads not taken.
- Focuses solely on approaches that allow the implementation to reconsider pointer invalidity only at specific well-marked points in the source code.

P2414R1 captures email-reflector discussions:

- Adds a summary of the requested changes to the abstract.
- Adds a forward reference to detailed expositions for atomics and volatiles to the "What We Are Asking For" section.
- Add a function `atomic_usable_ref` and change `usable_ptr::ref` to `usable_ref`. Change A2, A3, and Appendix A accordingly.
- Rewrite of section B5 for clarity.

P2414R0 extracts and builds upon the solutions sections from P1726R5 and P2188R1. Please see P1726R5 for discussion of the relevant portions of the standard, rationales for current pointer-zap semantics, expositions of prominent susceptible algorithms, the relationship between pointer zap and both happens-before and value-representation access, and historical discussions of options to handle pointer zap.

The WG14 C-Language counterparts to this paper, N2369 and N2443, have been presented at the 2019 London and Ithaca meetings, respectively.

Appendix: Name-Selection Guide

This appendix lists alternative names for `usable_ptr<T>` and `make_usable_ptr()`.

TL;DR: `ptr_bits<T>` and `launder_ptr_bits()` were chosen, following Anthony Williams's P2188R1 ("Zap the Zap: Pointers are sometimes just bags of bits"). There is still some contention on these names.

Please see P3580R0 ("The Naming of Things") for good advice on name selection.

Evaluation Criteria

From an email by Timur Doumler:

- List naming design goals.
- Sort goals by priority.
- List all proposed names.
- Determine which names best satisfy the design goals.

Naming Design Goals

These are in rough descending priority order.

- Only the value representation is preserved. Any implicit provenance is lost. However, any explicit provenance represented within the value representation (for example, that of ARM MTE) is preserved.
- When converted back to a pointer, implicit provenance might be regenerated from some object that is in existence at about that time. In painful detail:
    - Only those objects whose storage-duration beginning happened before or happened concurrently with the conversion back to a pointer need be considered.
    - Of those objects, if there is a set whose members that cause subsequent use of the corresponding pointers have defined behavior, then an object from that set must be selected.
    - As a consequence, any object whose storage-duration beginning happens after the conversion back to a pointer need ***not*** be considered by a C++ implementation.
- The implicit provenance that is regenerated upon conversion back to a pointer might not have any relation to that which was discarded when the pointer was converted to value-representation normal form.

Name Classification

Reject overly long names, which are shown in red below.

Reject names that are insufficiently clear, which are shown in yellow below.

List of Proposed Names

These were also put forward for `bag_of_bits_ptr<T>`, which was maligned as being too much of an inside joke:

- `ptr_bits<T>`: (Daveed Vandevoorde <daveed@edg.com>)
- `pointer_bits<T>`: (Daveed Vandevoorde <daveed@edg.com>, Peter Dimov <pdimov@gmail.com>), but the "`pointer_`" has no expressive advantage over "`ptr_`" and is longer.
- `pointer_value<T>`: (Tony V E <tvaneerd@gmail.com>), but this does not distinguish from the full pointer value, implicit provenance and all.
- `potential_pointer<T>`: (Tony V E <tvaneerd@gmail.com>), which has great meaning to those who have fully internalized P2434R4 ("Nondeterministic pointer provenance"), but might not help the larger group who have not.
- `pob_ptr<T>`: (Matt Bentley <mattreecebentley@gmail.com>), but the expansion to "plain old bits" is not likely to occur to many people.
- `rawbits_pointer<T>`: (Matt Bentley <mattreecebentley@gmail.com>), but this has no expressive advantage over `ptr_bits<T>` and is longer.
- `representation_pointer<T>`: (Tom Honermann <tom@honermann.net>), but overly long.
- `pointer_representation<T>`: (Alisdair Meredith <alisdairm@me.com>), but overly long.

- `rep_pointer<T>`: (Tom Honermann <tom@honermann.net>), but there are too many things that “rep” might expand to.
- `representation_ptr<T>`: (Tom Honermann <tom@honermann.net>)
- `rep_ptr<T>`: (Tom Honermann <tom@honermann.net>), but there are too many things that “rep” might expand to.
- `bag_of_ptr_bits<T>`: (Ville Voutilainen <ville.voutilainen@gmail.com>), but might not help those who are unaware of P2188R1 (“Zap the Zap: Pointers are sometimes just bags of bits”).
- `value_representation_pointer<T>`: But overly long.
- `value_representation_ptr<T>`: But overly long.
- `value_rep_ptr<T>`:

This leaves `ptr_bits<T>` and `value_rep_ptr<T>`. Although `value_rep_ptr<T>` is more closely related to the wording of the standard, this wording has been subject to recent change, so `ptr_bits<T>` is preferred.

A `launder_` prefix will denote the free function.

The result is a class named `ptr_bits<T>` and a free function named `launder_ptr_bits<T>`.

Historical: Selecting a Name for `usable_ptr<T>`

Although this name is very clear to those of us who have been doing pointer-zap, it does leave open the question “usable for what???”. Here is a list of alternatives, with the choice at that time highlighted in green:

- `prospective_ptr<T>`: (Fabio Fracassi <f.fracassi@gmx.net>)
    - + Says what it relates to.
    - - Might not mean much to those unfamiliar with this issue.
    - - Also is misleading on systems such as ARM MTE that reify some provenance bits.
- `provenance_ptr<T>`: (Ville Voutilainen <ville.voutilainen@gmail.com> and Alisdair Meredith <alisdairm@me.com>)
    - + Says what it relates to.
    - - Might not mean much to those unfamiliar with this issue.
    - - Also is misleading on systems such as ARM MTE that reify provenance.
- `unzappable_ptr<T>`:
    - + Easily understood.
    - - Not as accurate as one might want.
    - - The term “zap” is fun, but does not connect to any formal terminology.
- `zap_immune_ptr<T>`:
    - + Easily understood.
    - - Not as accurate as one might want.
    - - The term “zap” is fun, but does not connect to any formal terminology.
- `valid_ptr<T>`: (Matthew Bentley <mattreecebentley@gmail.com>)
    - + Easily understood.
    - - Not all that accurate because if you read out a `valid_ptr<T>` before the allocation, the pointer is invalid.
- `comparable_ptr<T>`: (Bronek Kozicki <brok@incorrekt.com>)

    - + Easily understood.
    - - Does not indicate the motivating use case, namely ABA-tolerant concurrent algorithms.
- `address_ptr<T>`: (Davis Herring <herring@lanl.gov>)
    - + Easily understood.
    - + Explains what is really happening in many implementations.
    - - Inaccurate for implementations (such as ARM MTE) that reify some provenance bits.
- `bag_of_bits_ptr<T>`: (Ville Voutilainen <ville.voutilainen@gmail.com> and Paul E. McKenney <paulmck@kernel.org>)
    - + Easily understood, particularly with reference to P2188R1 ("Zap the Zap: Pointers are sometimes just bags of bits").
    - + Explains what is happening on all implementations.
    - + Is not making provenance or validity promises that it cannot keep.
- Your ideas here! But it is too late, because `bag_of_bits_ptr<T>` beat you to it.

The following summarizes `bag_of_bits_ptr<T>` semantics:

> It is just a bag of bits.
> And a pointer to type T.
> It's untyped just right now.
> And a T* it soon will be!
>
> But if you do an allocation,
> And convert to T* some time before
> Then head right to the train station.
> For all meaning's out the door!

Historical: Selecting a Name for `make_usable_ptr()`

There are a number of issues with the name `make_usable_ptr()`, perhaps first and foremost that it returns a `T*` rather than a `usable_ptr<T>`. Here is a list of alternatives, with the current choice highlighted in green:

- `launder_prospective()`:
    - + Leverages the name of the existing `std::launder()` function.
    - + Extends the `std::launder()` function by accepting arguments that are invalid pointers.
    - + Allows the `std::launder()` function to continue to diagnose the passing of invalid pointers.
    - - Inaccurate for implementations (such as ARM MTE) that reify some provenance bits.
- `launder_invalid_pointer()`: see `launder_prospective()`. (Nic <n.morales.0@gmail.com>)
- `launder_invptr()`: see `launder_prospective()`.
- `launder_ptr_provenance()`: see `launder_prospective()`.
- `launder_valid_pointer()`: (Matthew Bentley <mattreecebentley@gmail.com>)
    - + Leverages the name of the existing `std::launder()` function.
    - + Extends the `std::launder()` function by accepting arguments that are invalid pointers.
    - + Allows the `std::launder()` function to continue to diagnose the passing of invalid pointers.
    - - Inaccurate for implementations (such as ARM MTE) that reify some provenance bits, which this function will not launder.

    - - Could confuse people into thinking that it has effect only on valid pointers.
- `make_raw_usable_ptr()`: (Nico Josuttis <nico@josuttis.de>)
    - + Says what it does.
    - + Notes the raw-pointer nature of the API.
    - - Loses analogy with `std::launder()`.
    - - Only makes the pointer usable if it does not happen before the allocation that provides validity.
- `prospective_provenance_ptr()`:
    - + Says what it does.
    - - Loses analogy with `std::launder()`.
    - - Inaccurate for implementations (such as ARM MTE) that reify some provenance bits, which this function does not affect.
- `revalidate_pointer()`:
    - + Says what it does.
    - - Loses analogy with `std::launder()`.
    - - Inaccurate for implementations (such as ARM MTE) that reify some provenance bits, which this function does not affect.
- `reevaluate_provenance_ptr()`:
    - +/- Kind of says what it does.
    - - Loses analogy with `std::launder()`.
    - - Inaccurate for implementations (such as ARM MTE) that reify some provenance bits, which this function does not affect.
- `regenerate_provenance_ptr()`:
    - +/- Kind of says what it does.
    - - Loses analogy with `std::launder()`.
    - - Inaccurate for implementations (such as ARM MTE) that reify some provenance bits, which this function does not affect.
- `recompute_provenance()`: (Ville Voutilainen <ville.voutilainen@gmail.com> and Alisdair Meredith <alisdairm@me.com>)
    - +/- Kind of says what it does.
    - - Loses analogy with `std::launder()`.
    - - Inaccurate for implementations (such as ARM MTE) that reify some provenance bits, which this function does not affect.
- `recompute_ptr_provenance()`: (Fabio Fracassi <f.fracassi@gmx.net>)
    - +/- Kind of says what it does.
    - -/- Misleading for implementations (such as ARM MTE) that reify some provenance bits, which this function does not affect.
    - - Loses analogy with `std::launder()`.
- `update_provenance_ptr()`:
    - +/- Kind of says what it does.
    - - Loses analogy with `std::launder()`.
- `ptr_immune_to_zap()`:
    - - Overstates the case.
    - - The term “zap” is fun, but does not connect to any formal terminology.
- `make_comparable_ptr<T>`: (Bronek Kozicki <brok@incorrekt.com>)
    - + Easily understood.
    - - Does not indicate the most important use case, namely ABA-tolerant concurrent algorithms.

- `launder_bag_of_bits_ptr()`:
    - + Retains the connection to laundering.
    - + Retains the connection to `bag_of_bits_ptr<T>`.
    - + Says what it really does: Launders the pointer from a bag-of-bits perspective, thus easily understood, especially with reference to P2188R1 ("Zap the Zap: Pointers are sometimes just bags of bits").
    - + Avoids confusion for implementations such as ARM MTE that have some provenance information in the actual in-memory representation.
- Your ideas here!

But what is in a name? But it is too late, because `launder_bag_of_bits_ptr()` beat you to it.