Pointer tagging for custom pointer types

Document #:	P4358R0
Date:	2026-09-21
Programming Language C++	
Audience:	LEWG

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

This paper proposes support for pointer-like types in `pointer_tag_pair`.

Motivation

The recently added `pointer_tag_pair` requires the pointer type to be an actual pointer, and does not support pointer-like user-defined types. We propose a customization mechanism (`taggable_pointer_traits`) to support such types.

The motivational use cases are as follows:

Opaque pointer types

Opaque pointers, i.e., classes that wrap a pointer in a type to block direct access to the pointer, or to achieve some form of strong typing, are fairly common. And while their alignment bits are free, they are not accessible without the proposed `taggable_pointer_traits`.

Note that it is not generally possible to use a smart, fancy, or owning pointer with a non-trivial constructor or destructor with this (or similar) interface.

Nested pointer-tag pair

Another use case, used a lot in LLVM, is to nest `pointer_tag_pair`. For example, it is possible to define `pointer_tag_pair<pointer_tag_pair<int*, 1, unsigned>, 1, unsigned>`. LLVM has complex structures and memory comes at a premium, so having to resort to that sort of optimization is fairly common. This paper proposes a specialization of `taggable_pointer_traits` for `pointer_tag_pair` that makes this possible.

Function pointers

While this goes a bit beyond what can be portably and conformingly achieved, `pointer_tag_pair` could be user-specialized to allow storing a function pointer inside a `pointer_tag_pair`.

Other uses for `taggable_pointer_traits`

LLVM uses its equivalent of `taggable_pointer_traits` to store a discriminated union of pointer types in a single `void*`. Pointer-sized discriminated variants are a fairly common utility (Qt and Folly have similar tools, for example). While we are not proposing such a type, `taggable_pointer_traits` can be reused for that use case (or any other use case that wants to query alignment information of pointer-like types).

Design

A trait, `taggable_pointer_traits` (for lack of a better name), offers the following interface:

```
template <>
struct std::taggable_pointer_traits<T> {
    static constexpr unsigned bits_available = /*...*/;
    static constexpr void* to_void_pointer(T p) noexcept;
    static constexpr T from_void_pointer(void* v) noexcept;
};
```

The standard provides a specialization for `pointer_tag_pair` and `T*` (the primary template is not defined).

The trait is queried by `pointer_tag_pair`. In LLVM, the trait is a template parameter (like, e.g., `char_traits` in string), which offers another layer of customization, which LLVM does not use, as far as I can tell. The only use case I can imagine would be to customize the behavior for `void*` — which the standard considers to have an alignment of `0`, but it is sometimes useful to assume a greater alignment depending on the use case and allocator. However, a custom opaque pointer type over `void*`, or the use of `from_overaligned`, would cover this use case. Generally, any scenario where one would want to customize the behavior of a given pointer type can be achieved by wrapping it in an opaque pointer type.

There are a few notable changes to the status quo:

- To make nested `pointer_tag_pair` work, we need to specify how the tag is stored rather than leave it implementation-defined. I.e., the tag is stored in the most significant of the available least-significant (alignment) bits.

- `pointer_tag_pair::element_type` is removed, as it's unclear that all opaque pointer types could or would want to expose that information.
- The `constexprness` of Hana's design remains; however, user-defined specializations of the trait need not be `constexpr`.

Open questions

- The current design mandates that the pointer type be default-constructible. We could instead make `pointer_tag_pair` conditionally default-constructible.
- I think we could provide a specialization of `taggable_pointer_traits` for `coroutine_handle`.

What this is not

Despite this paper extending `pointer_tag_pair` to non-pointer types, that utility still remains very much built around the idea of reusing pointer-alignment bits. Some early feedback on this paper expressed a desire for more unused-bits optimizations, including

- Tombstone values in `std::optional`.
- Recycling of padding bits.
- Owning pointer/tag pair types (I'm not aware of anyone doing that?)
- Pointer compression
- etc.

This proposal is none of these things. Each of these features requires a different design or controlling trait that the interface of `pointer_tag_pair` or `taggable_pointer_traits` cannot accommodate.

Implementation

An implementation, based on Hana Dusíková's initial implementation of P3125R6 [P3125R6], is available on Compiler Explorer. By virtue of it being a pure-library implementation, it does not support using `pointer_tag_pair` as a constant expression.

Wording

🚧 Header <memory> synopsis [memory.syn]

Modify **[memory.syn]** as indicated:

```
namespace std {
  // [...]

  // [ptrtag], pointer tagging
  inline constexpr unsigned max_pointer_bits_available = see below;                //
    freestanding
  constexpr unsigned pointer_bits_available(size_t alignment);                     //
    freestanding
```

```
  template<class Ptr> struct taggable_pointer_traits; // freestanding

  template<class T>
  constexpr unsigned bits-available = pointer_bits_available(alignof(T)); // exposition only

  template<class U> using element-of = pointer_traits<U>::element_type;// exposition only

  template<class Ptr, unsigned BitsRequested =
    bits-available<element-of<Ptr>>taggable_pointer_traits<Ptr>::bits_available,
          class TagT = unsigned>
    class pointer_tag_pair;                                                    //
    freestanding

  // [...]
}
```

🔲 Pointer tagging [ptrtag]

Insert a new subclause after **[ptrtag.bits]**:

🔲 Taggable pointer traits [ptrtag.traits]

🔲 General [ptrtag.traits.general]

`taggable_pointer_traits` provides an interface allowing the use of a pointer-like type with `pointer_tag_pair`.

```
namespace std {
  template<class Ptr> struct taggable_pointer_traits;          // freestanding
}
```

A program may specialize `taggable_pointer_traits` for a program-defined pointer type [namespace.std].

[*Note:* An operation of a `pointer_tag_pair` is usable in constant expressions only if the traits operations are constant expressions. — *end note*]

🔲 `taggable_pointer_traits` specializations [ptrtag.traits.specializations]

Let *`COPY_CONST`*`(A, B)` be `const B` if `A` is a const type, otherwise `B`.

```
namespace std {
  template<class T> struct taggable_pointer_traits<T*> {       // freestanding
    static constexpr unsigned bits_available = see below;

    static constexpr COPY_CONST(T, void)* to_void_pointer(T* p) noexcept;
    static constexpr T* from_void_pointer(COPY_CONST(T, void)* v) noexcept;
  };
}
```

`static constexpr unsigned bits_available = see below;`

`bits_available` is `0` if `is_void_v<T>` is `true`, and `pointer_bits_available(alignof(T))` otherwise.

`static constexpr COPY_CONST(T, void)* to_void_pointer(T* p) noexcept;`

Returns: `p`.

`static constexpr T* from_void_pointer(COPY_CONST(T, void)* v) noexcept;`

Returns: `static_cast<T*>(v)`.

🔲 Class template `pointer_tag_pair` [ptrtag.pair]

🔲 General [ptrtag.pair.general]

Modify **[ptrtag.pair.general]** as indicated:

```
namespace std {
  template<class U, class PtrT, unsigned BitsRequested, size_t Alignment = alignof(U)>
    concept tagging-compatible-pointee =                          // exposition only
      is_pointer_v<PtrT> && is_object_v<U> &&
      convertible_to<U*, PtrT> &&
      pointer_bits_available(Alignment) >= BitsRequested &&
      (is_void_v<~~element-of~~remove_pointer_t<PtrT>> ||
       is_scalar_v<~~element-of~~remove_pointer_t<PtrT>> ||
       is_union_v<~~element-of~~remove_pointer_t<PtrT>> ||
       is_pointer_interconvertible_base_of_v<~~element-of~~remove_pointer_t<PtrT>, U>);

  template<class TagT>
    constexpr unsigned tag-bit-width(TagT value) noexcept;        // exposition only
  template<class TagT>
    constexpr uintptr_t tag-value(TagT value) noexcept;               // exposition only

  template<class Ptr, unsigned BitsRequested =
    ~~bits-available<element-of<Ptr>>~~taggable_pointer_traits<Ptr>::bits_available,
          class TagT = unsigned>
  class pointer_tag_pair {                                        // freestanding
  public:
    using pointer_type        = Ptr;
    using traits_type = taggable_pointer_traits<Ptr>;
    ~~using element_type = pointer_traits<Ptr>::element_type;~~
    using tagged_pointer_type = see below;
    using tag_type            = TagT;
    static constexpr unsigned bits_requested = BitsRequested;

    // [ptrtag.pair.cons], constructors
    constexpr pointer_tag_pair() noexcept;

    template<tagging-compatible-pointee<pointer_type, bits_requested> U>
      constexpr pointer_tag_pair(U* p, tag_type t);
```

```
    constexpr pointer_tag_pair(nullptr_t p, tag_type t);
    constexpr pointer_tag_pair(pointer_type p, tag_type t)
      requires (bits_requested <= traits_type::bits_available);

    // [...]

    // [ptrtag.pair.comp], comparisons
    friend constexpr see below operator<=>(pointer_tag_pair lhs, pointer_tag_pair rhs)
      noexcept(see below) requires three_way_comparable<pointer_type>
    && three_way_comparable<tag_type>;
    friend constexpr bool operator==(pointer_tag_pair, pointer_tag_pair)
      noexcept(see below) requires equality_comparable<pointer_type>
    && equality_comparable<tag_type>;
  };

  template<class Ptr>
    pointer_tag_pair(Ptr*)
      -> pointer_tag_pair<Ptr*>;
  template<class Ptr, class TagT>
    pointer_tag_pair(Ptr*, TagT tag)
      -> pointer_tag_pair<Ptr*,
    bits-available<element-of<Ptr>>taggable_pointer_traits<Ptr>::bits_available, TagT>;
}
```

Mandates:

- `is_same_v<remove_cvref_t<Ptr>, Ptr>` is `true`.
- `is_same_v<remove_cvref_t<TagT>, TagT>` is `true`.
- ~~`is_pointer_v<Ptr> && !is_function_v<remove_pointer_t<Ptr>` is `true`.~~
- `traits_type::bits_available <= max_pointer_bits_available` is `true`.
- `is_same_v<remove_const_t<remove_pointer_t<tagged_pointer_type>>, void>` is `true`.
- `is_same_v<decltype(traits_type::from_void_pointer(declval<tagged_pointer_type>())), Ptr>` is `true`.
- `is_default_constructible_v<Ptr>` is `true`.
- `is_trivially_copyable_v<Ptr>` is `true`.
- `sizeof(Ptr) == sizeof(uintptr_t) && alignof(Ptr) == alignof(uintptr_t)` is `true`.
- `noexcept(traits_type::to_void_pointer(declval<Ptr>()))` is `true`.
- `noexcept(traits_type::from_void_pointer(declval<tagged_pointer_type>()))` is `true`.
- `is_unsigned_v<UT>` is `true`, where `UT` is `underlying_type_t<TagT>` if `TagT` is an enumeration type, and `TagT` otherwise.
- ~~`sizeof(TagT) <= sizeof(void*)` is `true`.~~
- `BitsRequested <= max_pointer_bits_available` is `true`.

Add to **[ptrtag.pair.general]**:

Let *tag-shift* and *tag-mask* be as follows:

```
static constexpr unsigned tag-shift =  // exposition only
  traits_type::bits_available > bits_requested ? traits_type::bits_available - bits_requested
     : 0;
static constexpr uintptr_t tag-mask =   // exposition only
  ((uintptr_t(1) << bits_requested) - 1) << tag-shift;
```

[*Note:* The tag occupies the most significant of the bits reported by `traits_type::bits_available`, which leaves the *tag-shift* least significant bits of the representation unused. — *end note*]

```
template<class TagT> constexpr uintptr_t tag-value(TagT value) noexcept;
```

Returns: `static_cast<uintptr_t>(to_underlying(value))` if `is_enum_v<TagT>` is `true`, otherwise `static_cast<uintptr_t>(value)`.

🔲 Constructors [ptrtag.pair.cons]

Modify **[ptrtag.pair.cons]** as indicated:

```
constexpr pointer_tag_pair() noexcept;
```

Postconditions: `pointer()` is equal to ~~`nullptr`~~`pointer_type()` and `tag()` is equal to `TagT()`.

[*Editor's note:* [...]]

Add to **[ptrtag.pair.cons]**:

```
constexpr pointer_tag_pair(pointer_type p, tag_type t)
  requires (bits_requested <= traits_type::bits_available);
```

Constant When: Preconditions are met and `traits_type::to_void_pointer(p)` is a constant expression.

Preconditions:

- `p` is not a pointer past the end of an object [basic.compound].
- The `bits_requested` least significant bits of `traits_type::to_void_pointer(p)` are zero.
- *tag-bit-width*`(t) <= bits_requested` is `true`.

Postconditions: `pointer()` is equal to `p` and `tag()` is equal to `t`.

Throws: Nothing.

🔲 Tagged pointer operations [ptrtag.pair.tagops]

Modify **[ptrtag.pair.tagops]** as indicated:

```
using tagged_pointer_type = see below;
```

`tagged_pointer_type` denotes *cv* `void*`, for *cv* such that pointer denotes *cv* `U*` for some type `U`. `tagged_pointer_type` denotes the type of `traits_type::to_void_pointer(p)` for a value `p` of type `pointer_type`.

```
tagged_pointer_type tagged_pointer() const noexcept;
```

Returns: An unspecified value `tp` of `tagged_pointer_type` type such that for any specialization `DP` of `pointer_tag_pair` and alignment value `A` for which

- `pointer_bits_available(A) >= DP::bits_requested` is `true`,
- `is_sufficiently_aligned<A>(ptr)` is `true`, and
- *`tag-bit-width`*`(tag) <= DP::bits_requested` is `true`,

`DP::from_tagged(tp)` produces an object `dp` such that `reinterpret_cast<pointer_type>(dp.pointer())` is equal to `ptr` and `static_cast<TagT>(dp.tag())` is equal to `tag`.

Returns: A value `tp` of type `tagged_pointer_type` for which the following expression is `true`:

```
bit_cast<uintptr_t>(tp) ==
  ((bit_cast<uintptr_t>(traits_type::to_void_pointer(pointer())) & ~tag-mask) |
   (tag-value(tag()) << tag-shift))
```

[*Note:* This function returns an invalid pointer value [basic.compound] with implementation-defined behavior [ptrtag.bits]. — *end note*]

```
static pointer_tag_pair from_tagged(tagged_pointer_type p) noexcept;
```

Returns: An object `dp` of type `pointer_tag_type`, such that

- `reinterpret_cast<pointer_type>(sp.pointer())` is equal to `dp.pointer()` and `static_cast<TagT>(sp.tag())` is equal to `dp.tag()`, if `p` is equal to `sp.tagged_pointer()` for some object `sp` of a type that is a specialization of `pointer_tag_type`, such that for some alignment value `A`
  - `pointer_bits_available(A) >= bits_requested` is `true`,
  - `is_sufficiently_aligned<A>(sp.pointer())` is `true`, and
  - *`tag-bit-width`*`(sp.tag()) <= bits_requested` is `true`,
- otherwise, the values of `dp.pointer()` and `dp.tag()` are unspecified.

Returns: An object `ptp` of type `pointer_tag_pair` such that

- `ptp.pointer()` is equal to `traits_type::from_void_pointer(SP::traits_type::to_void_pointer(sp.pointer()))` and `ptp.tag()` is equal to `static_cast<tag_type>(sp.tag())`, if `p` is equal to `sp.tagged_pointer()` for some object `sp` of a type `SP` that is a specialization of `pointer_tag_pair` for which
  - `SP::`*`tag-shift`* is equal to *`tag-shift`*, and

- *tag-bit-width*(sp.tag()) <= bits_requested is true;

- otherwise, the values of ptp.pointer() and ptp.tag() are unspecified.

🔲 Comparisons [ptrtag.pair.comp]

Modify **[ptrtag.pair.comp]** as indicated:

```
friend constexpr auto operator<=>(pointer_tag_pair lhs, pointer_tag_pair rhs)
  noexcept(see below) requires three_way_comparable<pointer_type>
    && three_way_comparable<tag_type>;
```

Effects: Equivalent to:

```
return pair(lhs.pointer(), lhs.tag()) <=> pair(rhs.pointer(), rhs.tag());
```

Remarks: The exception specification is equivalent to noexcept(pair(lhs.pointer(), lhs.tag()) <=> pair(rhs.pointer(), rhs.tag())).

Recommended practice: Directly compare the internal bit representations if the expression lhs.tag() <=> rhs.tag() results in a call to a built-in operator <=> comparing values of type tag_type, and the expression lhs.pointer() <=> rhs.pointer() results in a call to a built-in operator <=> comparing values of type pointer_type.

[*Note:* The result of comparing unrelated pointers is unspecified. — *end note*]

```
friend constexpr bool operator==(pointer_tag_pair lhs, pointer_tag_pair rhs)
  noexcept(see below) requires equality_comparable<pointer_type>
    && equality_comparable<tag_type>;
```

Effects: Equivalent to:

```
return pair(lhs.pointer(), lhs.tag()) == pair(rhs.pointer(), rhs.tag());
```

Remarks: The exception specification is equivalent to noexcept(pair(lhs.pointer(), lhs.tag()) == pair(rhs.pointer(), rhs.tag())).

Recommended practice: Directly compare the internal bit representations if the expression lhs.tag() == rhs.tag() results in a call to a built-in operator == comparing values of type tag_type, and the expression lhs.pointer() == rhs.pointer() results in a call to a built-in operator == comparing values of type pointer_type.

🔲 Taggable pointer traits for `pointer_tag_pair` [ptrtag.pair.traits]

Add a new subclause at the end of **[ptrtag.pair]**:

```
namespace std {
  template<class Ptr, unsigned BitsRequested, class TagT>
  struct taggable_pointer_traits<                              // freestanding
    pointer_tag_pair<Ptr, BitsRequested, TagT>> {
    static constexpr unsigned bits_available = see below;
```

```
    static pointer_tag_pair<Ptr, BitsRequested, TagT>::tagged_pointer_type
      to_void_pointer(pointer_tag_pair<Ptr, BitsRequested, TagT> p) noexcept;
    static pointer_tag_pair<Ptr, BitsRequested, TagT> from_void_pointer(
      pointer_tag_pair<Ptr, BitsRequested, TagT>::tagged_pointer_type v) noexcept;
  };
}
```

Let `PTP` denote `pointer_tag_pair<Ptr, BitsRequested, TagT>`.

```
static constexpr unsigned bits_available = see below;
```

`bits_available` is `PTP::tag-shift` [ptrtag.pair.general].

```
static PTP::tagged_pointer_type to_void_pointer(PTP p) noexcept;
```

Returns: `p.tagged_pointer()`.

```
static PTP from_void_pointer(PTP::tagged_pointer_type v) noexcept;
```

Returns: `PTP::from_tagged(v)`.

Feature-test macro

[*Editor's note:* Bump `__cpp_lib_pointer_tag_pair` to the date of adoption.]

Acknowledgments

Thanks to Hana Dusíková for her work on P3125R6, and for leaving the design open to the extension proposed here.

References

[P3125R6] Hana Dusíková *constexpr pointer tagging*
https://wg21.link/P3125R6

[LLVMPointerIntPair] The LLVM Project *llvm::PointerIntPair Class Template Reference*
https://llvm.org/doxygen/classllvm_1_1PointerIntPair.html

[LLVM94324] Nikolas Klauser *[libc++] Add __pointer_int_pair*
https://github.com/llvm/llvm-project/pull/94324

[Tiny] Jonathan Müller *tiny: `pointer_variant_impl` and `pointer_tiny_storage`*
https://github.com/foonathan/tiny

[QBiPointer] The Qt Project *`QBiPointer`*
https://github.com/qt/qtdeclarative/blob/dev/src/qml/qml/ftw/qbipointer_p.h

[DiscriminatedPtr] Meta *`folly::DiscriminatedPtr`*
https://github.com/facebook/folly/blob/main/folly/DiscriminatedPtr.h