`protocol`: Structural Subtyping for C++

ISO/IEC JTC1 SC22 WG21 Programming Language C++

P4148R1

Working Group: Library Evolution, Library

Date: 2026-05-12

Jonathan Coe <jonathanbcoe@gmail.com>

Hana Dusikova <hanicka@hanicka.net>

Antony Peacock <ant.peacock@gmail.com>

Philip Craig <philip@pobox.com>

Neelofer Banglawala <dr.nbanglawala@gmail.com>

Abstract

We propose `protocol<T, A>` and `protocol_view<T>`, standard library vocabulary types for structural subtyping in C++. Interfaces are specified as plain structs; any type whose member functions satisfy the interface is accepted without requiring explicit inheritance.

Any type that provides member functions with the same names and function signatures as those specified by the interface is considered to be *conforming* to the protocol.

The owning type, `protocol`, provides value semantics (const-propagation and deep-copies) and support for custom allocators for any conforming type.

The non-owning type, `protocol_view`, provides a lightweight reference to any conforming type, analogous to `std::span`.

Ideally both `protocol` and `protocol_view` would be generated by the compiler using static reflection, eliminating hand-written type-erasure boilerplate or custom build steps. This proposal assumes the availability of static reflection and code injection and focuses solely on the design of the class templates `protocol` and `protocol_view`.

History

Changes in revision R1

- Clarify special member function generation for `protocol` and `protocol_view`.
- Refine comparison with `std::proxy` (P3086).
- Add design alternatives section (relaxed structural subtyping, concepts, comparison operators).

- Add poll on reflection-based library features.

Changes in revision R0

- Initial revision.

Foreword

This is a very early stage design which we are sharing to further discussion of design differences with a series of competing proposals for structural-subtyping.

This paper explores a different approach to proxy and relies on reflection rather than templates for a smaller API surface.

Motivation

C++ is a multi-paradigm language, supporting object-oriented, generic, and functional programming styles. A key strength of the language is its ability to express different forms of polymorphism, allowing developers to select the most appropriate abstraction for a given context. However, this support is uneven: while some paradigms are directly supported by the language, others rely on idioms and library techniques.

One such case is dynamic structural polymorphism. While C++ provides strong support for static structural typing through concepts, it lacks a corresponding mechanism for runtime abstractions. In practice, this gap is addressed through the widespread use of type-erasure.

Standard library facilities such as `std::function`, `std::any`, `std::ranges::any_view` and the many other type-erasure based solutions demonstrate that the need for dynamic structural interfaces is both real and recurring. However, these solutions are implemented in an ad-hoc manner, requiring significant boilerplate and leading to inconsistent semantics across libraries.

This situation can be understood in terms of the broader polymorphism design space:

	Static	Dynamic
Nominal typing	Templates	Virtual
Structural typing	Concepts	—

The absence of a language-supported mechanism for dynamic structural typing explains the proliferation of type-erasure-based abstractions. Each such abstraction can be viewed as a manual encoding of a structural interface, tailored to a specific use case.

This paper proposes protocol types as a first-class library feature that fills this gap. Protocols unify and generalise existing type-erasure patterns, providing a consistent, non-intrusive mechanism for expressing dynamic structural polymorphism, while also providing consistent support for allocators:

	Static	Dynamic
Nominal typing	Templates	Virtual
Structural typing	Concepts	Protocol

Design

In C++26 we introduced `polymorphic<T>` which confers value-like semantics on a dynamically-allocated object. A `polymorphic<T>` may hold an object of a class publicly derived from `T`. In this proposal, we seek to further extend C++'s library of value-types with `protocol<T>` which can hold an object of any type so long as that type is a structural sub-type of `T`.

Like `polymorphic`, `protocol` supports deep-copies, const propagation and custom allocators. Like `polymorphic`, `protocol` has a valueless state after being moved from to allow move construction and move assignment without dynamic memory allocation.

Where `polymorphic<T>` is owning, `T*`, or `const T*` can be used as a non-owning reference type. There is no base class to take a pointer to for `protocol<T>` so we propose the addition of `protocol_view<T>` (and `protocol_view<const T>`) which are similar to `span` and `string_view` and give reference semantics to structural sub-types.

Generated structural subtyping

For a given struct, the corresponding `protocol` and `protocol_view` will implement all the public non-virtual, non-template member functions with identical constexpr, noexcept and const-qualification.

Unlike `polymorphic`, `protocol` and `protocol_view` do not provide `operator*` or `operator->` (or const-overloads) as there is no common base type to form a pointer or reference. Member functions from a `protocol` or `protocol_view` are generated so that the `protocol` or `protocol_view` is a valid structural subtype and can be called with traditional `instance.member_function(args)` syntax.

```
struct I {
    std::string func0(std::string_view) const noexcept;
    double func1(double);
    int func2(int);
    int func2(int, int); // Another overload, same name.
};
```

We then generate a partial template specialization for `protocol` and template specialization for `protocol_view`.

If the interface type `I` has deleted special member functions, then the corresponding special member functions for `protocol` will not be generated. For `protocol_view`, the copy constructor, move constructor, copy assignment and move assignment (and allocator-extended equivalents) are generated unconditionally.

```
template <typename Allocator>
class protocol<I, Allocator=std::allocator<void>> {

    // Default constructor.
    explicit constexpr protocol(); // conditionally-generated

    // Constructor from any conforming value.
    template <class U>
    constexpr explicit protocol(U&& u);

    // In-place constructor.
    template <class U, class... Ts>
    explicit constexpr protocol(std::in_place_type_t<U>, Ts&&... ts);

    // In-place constructor with initializer_list.
    template <class U, class J, class... Ts>
    explicit constexpr protocol(std::in_place_type_t<U>,
                                std::initializer_list<J> ilist, Ts&&... ts);

    // Copy constructor.
    constexpr protocol(const protocol& other);  // conditionally-generated

    // Move constructor.
    constexpr protocol(protocol&& other) noexcept;  // conditionally-generated

    // Allocator-extended default constructor.
    explicit constexpr protocol(std::allocator_arg_t,
                                const Allocator& alloc);  // conditionally-generated

    // Allocator-extended constructor from any conforming value.
    template <class U>
    constexpr explicit protocol(std::allocator_arg_t, const Allocator& alloc, U&& u);

    // Allocator-extended in-place constructor.
    template <class U, class... Ts>
    explicit constexpr protocol(std::allocator_arg_t, const Allocator& alloc,
                                std::in_place_type_t<U>, Ts&&... ts);
```

```
    // Allocator-extended in-place constructor with initializer_list.
    template <class U, class J, class... Ts>
    explicit constexpr protocol(std::allocator_arg_t, const Allocator& alloc,
                                std::in_place_type_t<U>,
                                std::initializer_list<J> ilist, Ts&&... ts);

    // Allocator-extended copy constructor.
    constexpr protocol(std::allocator_arg_t, const Allocator& alloc,
                       const protocol& other);  // conditionally-generated

    // Allocator-extended move constructor.
    constexpr protocol(std::allocator_arg_t, const Allocator& alloc,
                       protocol&& other) noexcept;  // conditionally-generated

    // Destructor.
    ~protocol();

    // structural-subtype (const and non-const) member functions.
    std::string func0(std::string_view) const noexcept;
    double func1(double) const;
    int func2(int);
    int func2(int, int); // Another overload, same name.

    // valueless after move
    constexpr bool valueless_after_move() const noexcept;
};

template <>
class protocol_view<I> {
    // Constructor from any mutable conforming object.
    template <typename T>
    constexpr protocol_view(T& obj) noexcept;

    // Construction from an rvalue conforming object is deleted.
    template <typename T>
    protocol_view(const T&&) = delete;

    // Constructor from a mutable protocol.
    template <typename Alloc>
    protocol_view(protocol<I, Alloc>& p) noexcept;

    // Construction from a protocol rvalue is deleted.
    template <typename Alloc>
    protocol_view(protocol<I, Alloc>&&) = delete;

    // structural-subtype (const and non-const) member functions.
```

```
    std::string func0(std::string_view) const noexcept;
    double func1(double) const;
    int func2(int);
    int func2(int, int); // Another overload, same name.
};

template <>
class protocol_view<const I> {
    // Constructor from any const conforming object.
    template <typename T>
    constexpr protocol_view(const T& obj) noexcept;

    // Construction from a const rvalue conforming object is deleted.
    template <typename T>
    protocol_view(const T&&) = delete;

    // Constructor from a const protocol.
    template <typename Alloc>
    protocol_view(const protocol<I, Alloc>& p) noexcept;

    // Construction from a const protocol rvalue is deleted.
    template <typename Alloc>
    protocol_view(const protocol<I, Alloc>&&) = delete;

    // Constructor from a mutable protocol.
    template <typename Alloc>
    protocol_view(protocol<I, Alloc>& p) noexcept;

    // Construction from a protocol rvalue is deleted.
    template <typename Alloc>
    protocol_view(protocol<I, Alloc>&&) = delete;

    // Constructor from a mutable protocol_view<I>.
    constexpr protocol_view(protocol_view<I> view) noexcept;

    // structural-subtype const member functions.
    std::string func0(std::string_view) const noexcept;
    double func1(double) const;
};
```

Code generation is currently implemented in a reference implementation with a custom build step but would be better implemented with generative reflection post C++26.

Function-like examples

We can use `protocol` and `protocol_view` with appropriate structural types to implement and extend the standard library's existing set of function-objects.

Consider the structural types below:

```
template <typename R, typename... Args>
struct Function {
    // All special member functions are defaulted.
    R operator()(Args&&... args) const;
};

template <typename R, typename... Args>
struct MoveOnlyFunction {
    // Deleted copy constructor and copy assignment.
    MoveOnlyFunction(const MoveOnlyFunction&) = delete;
    MoveOnlyFunction& operator=(const MoveOnlyFunction&) = delete;

    // Defaulted move constructor and move assignment.
    MoveOnlyFunction(MoveOnlyFunction&&) = default;
    MoveOnlyFunction& operator=(MoveOnlyFunction&&) = default;

    R operator()(Args&&... args) const;
};

template <typename R, typename... Args>
struct MutatingFunction {
    // All special member functions are defaulted.
    R operator()(Args&&... args);
};

template <typename R, typename... Args>
struct MoveOnlyMutatingFunction {
    // Deleted copy constructor and copy assignment.
    MoveOnlyMutatingFunction(const MoveOnlyMutatingFunction&) = delete;
    MoveOnlyMutatingFunction& operator=(const MoveOnlyMutatingFunction&) = delete;

    // Defaulted move constructor and move assignment.
    MoveOnlyMutatingFunction(MoveOnlyMutatingFunction&&) = default;
    MoveOnlyMutatingFunction& operator=(MoveOnlyMutatingFunction&&) = default;

    R operator()(Args&&... args);
};

struct OverloadedFunction {
    // All special member functions are defaulted.
    R1 operator()(Args1&&... args) const;
    R2 operator()(Args2&&... args);
```

```
    R3 operator()(Args3&&... args);
};
```

There is currently no function-type in the standard library that can represent an overload set. The table below is illustrative of how flexible `protocol` and `protocol_view` are:

Standard library type	Protocol equivalent
`copyable_function<R(Args...) const>`	`protocol<Function<R, Args...>>`
`move_only_function<R(Args...) const>`	`protocol<MoveOnlyFunction<R, Args...>>`
`function_ref<R(Args...) const>`	`protocol_view<Function<R, Args...>>`
`copyable_function<R(Args...)>`	`protocol<MutatingFunction<R, Args...>>`
`move_only_function<R(Args...)>`	`protocol<MoveOnlyMutatingFunction<R, Args...>>`
`function_ref<R(Args...)>`	`protocol_view<MutatingFunction<R, Args...>>`
???	`protocol<OverloadedFunction>`
???	`protocol_view<OverloadedFunction>`

Comparison with proxy

`proxy` (P3086, implemented in `ngcpp/proxy`) occupies an overlapping region of the design space: both proposals provide type-erased, non-intrusive runtime polymorphism without requiring inheritance.

Interface Definition `protocol` defines an interface as a C++ struct containing member-function declarations. The library (or compiler, given reflection) introspects the struct to synthesise a vtable and forwarding member functions. `proxy` requires the author to build a *Facade* explicitly using the `pro::facade_builder` template, combining dispatch objects such as `pro::member_dispatch` with `add_convention` calls. The `protocol` approach is unobtrusive: any existing struct, including those in third-party headers, can serve as an interface without modification. The `proxy` approach gives the author precise control over dispatch conventions but couples the interface definition to library implementation details.

Interaction syntax `protocol` synthesises member functions directly on the wrapper, so callers can call member functions directly: `p.draw()`. `proxy` requires indirection: `p->draw()`. Using `operator->` avoids potential name collisions with the erased type's methods; allowing direct member function calls makes a `protocol<T>` a drop-in structural substitute for any type conforming to `T`.

Layout constraints A `proxy` Facade encodes physical layout constraints directly in the type. This enables the compiler to apply `memcpy`-based relocation and to enforce specific memory budgets per interface. `protocol`, like `polymorphic` and `function`, does not prescribe any layout constraints and leaves details like small-buffer-optimization to be determined by implementers.

Subtype Substitution A `proxy<RichFacade>` can be implicitly converted to a `proxy<LeanFacade>` when `RichFacade` explicitly includes `LeanFacade` via `add_facade`. Because `protocol` interfaces are plain, independent structs with no declared relationship, the same zero-overhead conversion is not available.

Ownership Erasure `protocol` is uniquely owning, `protocol_view` is non-owning. `proxy` can store any suitable pointer-like object and offers a lifetime-independent interface where the lifetime of the pointer-like object is determined by the choice of pointer, not by `proxy`. `proxy_view` is, like `protocol_view`, non-owning.

Summary table The table below summarises the main design choices side by side.

Aspect	`protocol`	`proxy`
Interface definition	C++ struct	`facade_builder` + dispatch objects (explicit)
Interaction syntax	`p.draw()`	`p->draw()`
Layout constraints	Implementation defined	Encoded in the Facade type
Subtype substitution	Unsupported	Implicit via `add_facade`
Ownership model	Explicit	Erased

Design Alternatives

We discuss design alternatives that were considered and why they were not adopted.

Relaxed structural subtyping We require a type to have exactly matching function signatures as `I` to be considered a conforming type for `protocol<I>`. Implicit conversions *could* be allowed but this might lead to odd chains of implicit-conversion-led conformance where an object can be passed through a sequence of `protocol` (or `protocol_view`) objects to conform to the interface of the last `protocol`. Where implicit conversions are unidirectional this may lead to undesirable or surprising behaviour.

With some suitably compelling motivation, conformance via implicit conversions could be added to `protocol` in a later revision of the C++ standard without rendering existing code ill-formed.

Structure defined with concepts We use a struct rather than a concept to define the interface of the `protocol<I>` (and `protocol_view<I>`) specialization. A concept could be used but concepts are a more expert feature than is necessary to define a structural subtyping interface.

Internally, our reference implementation defines a concept from the interface struct to generate better compiler errors when non-conforming types are used.

Equality and comparison operators We do not generate equality or comparison operators. If the interface struct `I` in `protocol<I>` defines equality or comparison operators as inline friends or member functions, these are not generated for `protocol` or `protocol_view`.

Equality or comparison operators are not part of the core functionality of `protocol` or `protocol_view` but could be added in a later revision of the C++ standard.

Impact on the Standard

This proposal is a library extension. It requires language support for code injection from static reflection and the addition of a new standard library header `<protocol>`.

Polls

- Should we work to standardize `protocol` and `protocol_view`?
- Is implementing something *like* `protocol` and `protocol_view`, their design details aside, something that we would like C++ reflection to be able to do?

Reference Implementation

A reference implementation, using an AST-based Python code generator to simulate post-C++26 code injection, is available at https://github.com/jbcoe/cc-protocol. The implementation demonstrates the feasibility of vtable generation, allocator awareness, and the value semantics properties required by this proposal.

Acknowledgements

References

[PEP 544] *Protocols: Structural subtyping (static duck typing).* https://peps.python.org/pep-0544/

[P3019] *std::indirect and std::polymorphic*. https://isocpp.org/files/papers/P3019R14.pdf

[P2996] *Reflection for C++26*. https://isocpp.org/files/papers/P2996R13.html

[Metaclasses for generative C++] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p0707r5.pdf

[P3086R4 Proxy] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3086r5.html

[py_cppmodel] *Python wrappers for clang's parsing of C++ to simplify AST analysis*. https://github.com/jbcoe/py_cppmodel

Appendix A: Illustrative Implementation

This appendix provides an illustrative example of the proposed implementation using static reflection and code injection. Identifiers in UPPER CASE denote hypothetical reflection primitives or language features that are not currently part of the C++26 reflection proposal ([P2996R13]).

```
// Thoughts on how to use current and future reflection features for protocol.

#include <concepts>
#include <cstddef>
#include <memory>
#include <type_traits>
#include <utility>

namespace lifetime {

struct allocating_storage {
  void *ptr{nullptr};  // to make it default initializable

  constexpr allocating_storage() {}

  template <typename T>
  constexpr allocating_storage(T &&obj)
      : ptr{new std::remove_cvref_t<T>(std::forward<T>(obj))} {}

  template <typename T>
  constexpr void destroy_object(this allocating_storage &self) {
    delete static_cast<T *>(self.ptr);
  }

  template <typename T>
  constexpr auto &get_object(this auto &&self) {
    return std::forward_like<decltype(self)>(*static_cast<T *>(self.ptr));
```

```
  }
};

struct reference_storage {
  void *ptr;

  static reference_storage capture_object(auto &obj) {
    return std::addressof(obj);
  }

  template <typename T>
  constexpr auto &get_object(this auto &&self) {
    return *static_cast<T *>(self.ptr);
  }
};

template <size_t N, size_t Alignment>
struct short_buffer_storage {
  alignas(Alignment) std::array<char, N> data;

  short_buffer_storage(auto &&obj) : data{} {
    std::construct_at<std::remove_cvref_t<decltype(obj)>>(
        data.data(), std::forward<decltype(obj)>(obj));
  }

  static short_buffer_storage capture_object(auto &&obj) {
    return short_buffer_storage{std::forward<decltype(obj)>(obj)};
  }

  template <typename T>
  constexpr void release_object(this auto &&self) {
    delete std::start_lifetime_as<std::remove_reference_t<decltype(self)>>(
        self.data.data());
  }

  template <typename T>
  constexpr auto &get_object(this auto &&self) {
    return std::forward_like<decltype(self)>(
        *std::start_lifetime_as<std::remove_reference_t<decltype(self)>>(
            self.data.data()));
  }
};

}  // namespace lifetime

template <typename Source, typename Lifetime>
```

```
struct protocol_builder {
  struct wrapper;
  struct vtable_type;

  consteval {
    auto member_functions =
        members_of(^^Source) | std::views::filter(std::meta::is_function);

    // VTABLE
    auto vtable_members =
        wrapper_member_functions | transform([](std::meta::info mf) {
          auto params = parameters_of(mf);
          // this:
          const auto This = params[0];  // original this
          auto type = ^^Lifetime;
          if (is_const(This)) {
            type = add_const(type);
          }

          if (is_lvalue_reference_type(This)) {
            type = add_lvalue_reference(type);
          } else if (is_rvalue_reference_type(This)) {
            type = add_rvalue_reference(type);
          }

          params[0] = type;  // new `this`, but as first argument
          auto fptr_type = MAKE_FUNCTION_POINTER(
              {.return_type = return_type_of(mf),
               .parameters = params,
               .noexcept = is_noexcept(mf)}) return data_member_spec{
              .type = fptr_type, .name = identifier_of(mf)};
        }) |
        std::ranges::to<std::vector>;

    auto vtable = define_aggregate(^^vtable_type, vtable_members);

    // TODO: add copy / move / assign/ destroy support

    // TODO assignments needs special handling
    auto wrapper_member_functions =
        wrapper_member_functions | transform([](std::meta::info mf) {
          auto params = parameters_of(mf);
          // this:
          const auto This = params[0];  // original this
          auto type = ^^wrapper;
          if (is_const(This)) {
```

```
        type = add_const(type);
      }

      if (is_lvalue_reference_type(This)) {
        type = add_lvalue_reference(type);
      } else if (is_rvalue_reference_type(This)) {
        type = add_rvalue_reference(type);
      }

      params[0] = type;  // new `this`
      return MEMBER_FUNCTION_SPEC{.return_type = return_type_of(mf),
                                  .name = identifier_of(mf),
                                  .parameters = params};
    }) |
    std::ranges::to<std::vector>;

// wrapper will contain:
// pointer vtable + storage
std::vector<std::meta::info> wrapper_nonstatic_data_members{
    data_member_spec(add_pointer(add_const(vtable)), {.name = "__vtable"})};

if (is_default_constructible_type(^^Source)) {
  wrapper_member_functions.push_back(CONSTRUCTOR_SPEC{});
  wrapper_nonstatic_data_members.push_back(data_member_spec(
      ^^Lifetime, {
                      .name = "__storage", .DEFAULTED = true}));
} else {
  wrapper_nonstatic_data_members.push_back(data_member_spec(
      ^^Lifetime, {
                      .name = "__storage", .DEFAULTED = false}));
}

if (is_copy_constructible_type(^^Source)) {
  wrapper_member_functions.push_back(
      CONSTRUCTOR_SPEC{.type = add_lvalue_reference(add_const(^^Wrapper))});
}

if (is_copy_constructible_type(^^Source)) {
  wrapper_member_functions.push_back(
      CONSTRUCTOR_SPEC{.type = add_rvalue_reference(^^Wrapper)});
}

if (HAS_MEMBER_FUNCTION_TEMPLATE(^^Lifetime, "release_object")) {
  wrapper_member_functions.push_back(DESTRUCTOR_SPEC{});
}
```

```
    wrapper_member_functions.push_back(CONSTRUCTOR_SPEC{
        template constructor taking any compatible object with Source});

    // define class, but not only declares members!
    auto wrp = DEFINE_CLASS(^^wrapper, wrapper_member_functions,
                            wrapper_nonstatic_data_members);

    for (const auto member :
         member_of(wrp) | std::views::filter(std::meta::is_function)) {
      if (is_default_constructor(member)) {
        // nothing
      } else if (is_copy_constructor(member)) {
        DEFINE_CONSTRUCTOR(member, copy vtable pointer,
                           and call Lifetime.copy_object());
      } else if (is_move_constructor(member)) {
        DEFINE_CONSTRUCTOR(member, copy vtable pointer,
                           and call Lifetime.move_object());
      } else if (is_constructor(member)) {
        DEFINE_TEMPLATE_CONSTRUCTOR(
            member, pass auto &&object to storage,
            and assign vtable pointer to specialization);
      } else {
        // API from Source
      }
    }
  }
};

struct animal {
  void make_a_sound(float loudness) const;
};

template <typename T, typename Lifetime>
struct basic_protocol;

template <>
struct basic_protocol<animal, lifetime::allocating_storage> {
  using source_type = animal;
  using lifetime_type = lifetime::allocating_storage;

  lifetime_type __storage;

  struct __vtable_type {
    void (*__destroy_self)(lifetime_type &) = nullptr;
    lifetime_type (*__copy_self)(const lifetime_type &) = nullptr;
    lifetime_type (*__move_self)(lifetime_type &&) = nullptr;
```

```
  void (*make_a_sound)(const lifetime_type &, float loudness) = nullptr;
};

template <typename T>
static constexpr auto __vtable_implementation = __vtable_type{
    .__destroy_self =
        +[](lifetime_type &obj) -> void { obj.destroy_object<T>(); },
    .__copy_self = +[](const lifetime_type &obj) -> lifetime_type {
      return lifetime_type{obj.get_object<T>()};
    },
    .__move_self = +[](lifetime_type &&obj) -> lifetime_type {
      return lifetime_type{obj.get_object<T>()};
    },
    .make_a_sound = +[](const lifetime_type &obj, float loudness) -> void {
      obj.get_object<T>().make_a_sound(loudness);
    }};

const __vtable_type *__vtable{nullptr};

[[gnu::used]] basic_protocol() /*requires
                                  (std::default_initializable<source_type>)*/
    = default;

template <typename T>
basic_protocol(T &&object)
  requires(!std::same_as<basic_protocol, std::remove_cvref_t<T>>)
    : __storage{std::forward<T>(object)},
      __vtable{&__vtable_implementation<std::remove_cvref_t<T>>} {
  // constructs object in storage and sets the vtable
}

[[gnu::used]] basic_protocol(const basic_protocol &other)
    : __storage{other.__vtable->__copy_self(other.__storage)},
      __vtable{other.__vtable} {
  // asks the vtable to provide a copy of the storage
}

[[gnu::used]] basic_protocol(basic_protocol &&other)
    : __storage{other.__vtable->__move_self(std::move(other.__storage))},
      __vtable{other.__vtable} {
  // asks the vtable to provide a movecopy of the storage
}

[[gnu::used]] ~basic_protocol() { __vtable->__destroy_self(__storage); }

[[gnu::used]] void make_a_sound(float loudness) {
```

```
    return __vtable->make_a_sound(__storage, loudness);
  }
};

template <typename Source>
using protocol = basic_protocol<Source, lifetime::allocating_storage>;

#include <cstdio>

struct dog {
  void make_a_sound(float loudness) const { puts("bark"); }
};

protocol<animal> convert(dog &&d) { return {std::move(d)}; }
```