`indirect` and `polymorphic`: Vocabulary Types for Composite Class Design

ISO/IEC JTC1 SC22 WG21 Programming Language C++

P3019R14

Working Group: Library Evolution, Library

Date: 2025-02-10

Jonathan Coe <jonathanbcoe@gmail.com>

Antony Peacock <ant.peacock@gmail.com>

Sean Parent <sparent@adobe.com>

Abstract

We propose the addition of two new class templates to the C++ Standard Library: `indirect<T>` and `polymorphic<T>`.

Specializations of these class templates have value semantics and compose well with other standard library types (such as vector), allowing the compiler to correctly generate special member functions.

The class template `indirect` confers value-like semantics on a dynamically-allocated object. An `indirect` may hold an object of a class `T`. Copying the `indirect` will copy the object `T`. When an `indirect<T>` is accessed through a const access path, constness will propagate to the owned object.

The class template `polymorphic` confers value-like semantics on a dynamically-allocated object. A `polymorphic<T>` may hold an object of a class publicly derived from `T`. Copying the `polymorphic<T>` will copy the object of the derived type. When a `polymorphic<T>` is accessed through a const access path, constness will propagate to the owned object.

This proposal is a fusion of two earlier individual proposals, P1950 and P0201. The design of the two proposed class templates is sufficiently similar that they should not be considered in isolation.

History

Changes in R14

- Remove mandates that `T` is copy-constructible from `indirect`'s `operator=(U&&)`.
- Use `std::strong_ordering::less` instead of `false < true` in `operator <=>` for `indirect`.
- Amend `rhs` to `lhs` in valueless check for `operator<=>` for `indirect`.
- Add missing commas in wording.

Changes in R13

- Remove noexcept specification from `operator<=>` for `indirect`.
- Add a second clarifying example to tagged constructors explanatory text.
- Remove constraint `is_same_v<U, polymorphic>` is `false` on polymorphic constructors taking `in_place_type_t` and an intializer list.
- Order `polymorphic` constructor constraints consistently.
- Remove needless introduction of `UU` in `indirect` constraints.
- Update discussion of constraints and incomplete type support in appendix.

Changes in R12

- Fix `indirect` synopsis to include `explicit` on the default constructor.
- Replace "may only be X" with "may be X only" in specification of `indirect` and `polymorphic`.
- Change *constraints* on `T` where `T` could be an incomplete type to *mandates*.
- Remove *mandates* that `T` is a complete type where this is implicitly required by type_traits.
- `T` in `indirect` needs to be copy-constructible only for the copy constructor(s).
- Add discussion of constraints and incomplete type support in appendix.
- Fix specification of `<=>` to use `synth-three-way-result`.
- Change *constraints* on `operator==` for `indirect` to *mandates*.
- Remove constraints on `operator<=>` for `indirect`.
- Updates to non-technical specification sections to reflect design revisions for constraints and comparison.

Changes in R11

- Remove unnecesary `remove_const` from the specification of hash for indirect.
- Add a default template type parameter for single-argument constructors for indirect and polymorphic and for indirect's perfect-forwarding assignment.
- Add postconditions to say that the moved-from `indirect` is valueless in move assigment, move constructor and allocator-extended move construction. The same does not apply for polymorphic which permits a small buffer optimization.
- Add drafting note for use of italicised code font for exposition only variables.
- Prevent `T` from being `in_place_t` or a specialization of `in_place_type_t` for both indirect and polymorphic.
- Collect `in_place_t` and `in_place_type_t` constructors together.
- Define `UU` as `remove_cvref_t<U>` to simplify various requirements.
- Use `derived_from` rather than `is_base_of_v` in requirements for polymorphic.
- Require `is_same_v<remove_cvref_t<U>, U>` for `polymorphic` constructors taking `in_place_type_t<U>`.
- Check `is_same_v` constraints first.

Changes in R10

- Correct naming of explicit 'converting' constructors to 'single-argument' constructors.
- Amend naming of indirect's 'converting' constructor to 'perfect-forwarded' assignment.
- Correct changelog from R9.

Changes in R9

- Move throws clauses from individual constructor specifications to the start of constructors specification for indirect and polymorphic.
- Re-order constructors.
- Add perfect-forwarded assignment operator to `indirect`.
- Add single-argument constructors to `indirect` and `polymorphic`.
- Add intializer list constructors to `indirect` and `polymorphic`.
- Avoid use of 'heap' and 'free-store' in favour of 'dynamically-allocated storage'.

Changes in R8

- Wording cleanup in parallel with independent implementation.
- Add more explicit wording for use of `allocator_traits::construct` in `indirect` and `polymorphic` constructors.
- Prevent `indirect` and `polymorphic` classes from being instantiated with `in_place_t` and specializations of `in_place_type_t`.
- Strike mandates `T` is a complete type from indirect comparison operators and hash for consistency with reference wrapper.

Changes in R7

- Discuss `indirect`'s non-conditional copy constructor in the light of implementation tricks that would enable it.
- Improve wording for assignment operators to remove ambiguity.
- Add motivation for `valueless_after_move` member function.

Changes in R6

- Add `std::in_place_t` argument to indirect constructors.
- Amend wording for assignment operators to provide strong exception guarantee.
- Amend wording for swap to consider the valueless state.
- Remove comparison operators for `indirect` where they can be compiler-synthesized.
- Rename erroneous exposition only variable `allocator` to `alloc`.
- Add drafting note on exception guarantees behaviour to `swap`.

Changes in R5

- Fix wording for assignment operators to provide strong exception guarantee.
- Add missing wording for valueless hash.

Changes in R4

- Use constraints to require that the object owned by `indirect` is copy constructible. This ensures that `std::is_copy_constructible_v` does not give misleading results.
- Modify comparison of `indirect` allow comparsion of valueless objects. Comparisons are implemented in terms of `operator==` and `operator<=>` returning `bool` and `auto`.
- Remove `std::format` support for `std::indirect` as it cannot handle a valueless state.
- Allow copy, move, assign and swap of valueless objects, discuss similarities with variant.
- No longer specify constructors as uses-allocator constructing anything.
- Require `T` to satisfy the requirements of `Cpp17Destructible`.
- Rename exposition only variables `p_` to `p` and `allocator_` to `alloc`.
- Add discussion on incomplete types.
- Add discussion on explicit constructors.
- Add discussion on arithmetic operators and update change table.
- Remove references to `std::indirect`/`std::polymorphic` values terms under `[*.general]` sections.

Changes in R3

- Add explicit to constructors.
- Add constructor `indirect(U&& u, Us&&... us)` overload and requisite constraints.
- Add constructor `polymorphic(allocator_arg_t, const Allocator& a)` overload.
- Add discussion on similarities and differences with variant.
- Add table of breaking and non-breaking changes to appendix C.
- Add missing comparison operators and ensure they are all conditionally noexcept.
- Add argument deduction guides for `std::indirect`.
- Address incorrect `std::indirect` usage in composite example.
- Additions to acknowledgements.
- Address wording for `swap()` relating to `noexcept`.
- Address constraints wording for `std::indirect` comparison operators.
- Copy constructor now uses `allocator_traits::select_on_container_copy_construction`.
- Ensure swap and assign with self are nops.
- Move feature test macros to [version.syn].
- Remove `std::optional` specializations.
- Replace use of "erroneous" with "undefined behaviour".
- Strong exception guarantee for copy assignment.
- Specify constructors as uses-allocator constructing `T`.
- Wording review and additions to `<memory>` synopsis [memory.syn]

Changes in R2

- Add discussion on returning `auto` for `std::indirect` comparison operators.
- Add discussion of `emplace()` to appendix.
- Update wording to support allocator awareness.

Changes in R1

- Add feature-test macros.
- Add `std::format` support for `std::indirect`
- Add Appendix B before and after examples.
- Add preconditions checking for types are not valueless.
- Add constexpr support.
- Allow quality of implementation support for small buffer optimization for `polymorphic`.
- Extend wording for allocator support.
- Change *constraints* to *mandates* to enable support for incomplete types.
- Change pointer usage to use `allocator_traits` pointer.
- Remove `std::uses_allocator` specliazations.
- Remove `std::inplace_t` parameter in constructors for `std::indirect`.

- Fix `sizeof` error.

Motivation

The standard library has no vocabulary type for a dynamically-allocated object with value semantics. When designing a composite class, we may need an object to be stored indirectly to support incomplete types, reduce object size or support open-set polymorphism.

We propose the addition of two new class templates to the standard library to represent indirectly stored values: `indirect` and `polymorphic`. Both class templates represent dynamically-allocated objects with value-like semantics. `polymorphic<T>` can own any object of a type publicly derived from `T`, allowing composite classes to contain polymorphic components. We require the addition of two classes to avoid the cost of virtual dispatch (calling the copy constructor of a potentially derived-type object through type erasure) when copying of polymorphic objects is not needed.

Design requirements

We review the fundamental design requirements of `indirect` and `polymorphic` that make them suitable for composite class design.

Special member functions

Both class templates are suitable for use as members of composite classes where the compiler will generate special member functions. This means that the class templates should provide the special member functions where they are supported by the owned object type `T`. As `T` may be an incomplete type, the special member functions are unconditionally available to participate in overload resolution but would lead to an ill-formed program if they are called for a type that does not support them.

Deep copies

Copies of `indirect<T>` and `polymorphic<T>` should own copies of the owned object created with the copy constructor of the owned object. In the case of `polymorphic<T>`, this means that the copy should own a copy of a potentially derived type object created with the copy constructor of the derived type object.

Note: Including a `polymorphic` component in a composite class means that virtual dispatch will be used (through type erasure) in copying the `polymorphic` member. Where a composite class contains a polymorphic member from a known set of types, prefer `std::variant` or `indirect<std::variant>` if indirect storage is required.

`const` propagation

When composite objects contain `pointer`, `unique_ptr` or `shared_ptr` members they allow non-const access to their respective pointees when accessed through a const access path. This prevents the compiler from eliminating a source of const-correctness bugs and makes it difficult to reason about the const-correctness of a composite object.

Accessors of unique and shared pointers do not have const and non-const overloads:

```
T* unique_ptr<T>::operator->() const;
T& unique_ptr<T>::operator*() const;

T* shared_ptr<T>::operator->() const;
T& shared_ptr<T>::operator*() const;
```

When a parent object contains a member of type `indirect<T>` or `polymorphic<T>`, access to the owned object (of type `T`) through a const access path should be `const` qualified.

```
struct A {
    enum class Constness { CONST, NON_CONST };
    Constness foo() { return Constness::NON_CONST; }
    Constness foo() const { return Constness::CONST; }
};

class Composite {
    indirect<A> a_;

    Constness foo() { return a_->foo(); }
    Constness foo() const { return a_->foo(); }
};

int main() {
    Composite c;
    assert(c.foo() == A::Constness::NON_CONST);
    const Composite& cc = c;
    assert(cc.foo() == A::Constness::CONST);
}
```

Value semantics

Both `indirect` and `polymorphic` are value types whose owned object's storage is managed by the specified allocator.

When a value type is copied it gives rise to two independent objects that can be modified separately.

The owned object is part of the logical state of `indirect` and `polymorphic`. Operations on a const-qualified object do not make changes to the object's logical state nor to the logical state of owned objects.

The valueless state and interaction with `std::optional`

Both `indirect` and `polymorphic` have a valueless state that is used to implement move. The valueless state is not intended to be observable to the user. There is no `operator bool` or `has_value` member function. Accessing the value of an `indirect` or `polymorphic` after it has been moved from is undefined behaviour. We provide a `valueless_after_move` member function that returns `true` if an object is in a valueless state. This allows explicit checks for the valueless state in cases where it cannot be verified statically.

Without a valueless state, moving `indirect` or `polymorphic` would require allocation and moving from the owned object. This would be expensive and would require the owned object to be moveable. The existence of a valueless state allows move to be implemented cheaply without requiring the owned object to be moveable.

Where a nullable `indirect` or `polymorphic` is required, using `std::optional` is recommended. This may become common practice since `indirect` and `polymorphic` can replace smart pointers in composite classes, where they are currently used to (mis)represent component objects. Using dynamically-allocated storage for `T` should not make it nullable. Nullability must be explicitly opted into by using `std::optional<indirect<T>>` or `std::optional<polymorphic<T>>`.

Allocator support

Both `indirect` and `polymorphic` are allocator-aware types. They must be suitable for use in allocator-aware composite types and containers. Existing allocator-aware types in the standard, such as `vector` and `map`, take an allocator type as a template parameter, provide `allocator_type`, and have constructor overloads taking an additional `allocator_type_t` and allocator instance as arguments. As `indirect` and `polymorphic` need to work with, and in the same way, as existing allocator-aware types, they too take an allocator type as a template parameter, provide `allocator_type`, and have constructor overloads taking an additional `allocator_type_t` and allocator instance as arguments.

Modelled types

The class templates `indirect` and `polymorphic` have strong similarities to existing class templates. These similarities motivate much of the design; we aim for consistency with existing library types, not innovation.

Modelled types for `indirect`

The class template `indirect` owns an object of known type, permits copies, propagates const and is allocator aware.

- Like `optional` and `unique_ptr`, `indirect` can be in a valueless state; `indirect` can get into the valueless state only after being moved from, or after assignment or construction from a valueless state.
- `unique_ptr` and `optional` have preconditions for `operator->` and `operator*`: the behavior is undefined if `*this` does not contain a value.
- `unique_ptr` and `optional` mark `operator->` and `operator*` as noexcept: `indirect` does the same.
- `optional` and `indirect` know the underlying type of the owned object so can implement r-value qualified versions of `operator*`. For `unique_ptr`, the underlying type is not known (it could be an instance of a derived class) so r-value qualified versions of `operator*` are not provided.
- Like `vector`, `indirect` owns an object created by an allocator. The move constructor and move assignment operator for `vector` are conditionally noexcept on properties of the allocator. Thus for `indirect`, the move constructor and move assignment operator are conditionally noexcept on properties of the allocator. (Allocator instances may have different underlying memory resources; it is not possible for an allocator with one memory resource to delete an object in another memory resource. When allocators have different underlying memory resources, move necessitates the allocation of memory and cannot be marked noexcept.) Like `vector`, `indirect` marks member and non-member `swap` as noexcept and requires allocators to be equal.
- Like `optional`, `indirect` knows the type of the owned object so it can forward comparison operators and hash to the underlying object. A valueless `indirect`, like an empty `optional`, hashes to an implementation-defined value.

Modelled types for `polymorphic`

The class template `polymorphic` owns an object of unknown type, requires copies, propagates const and is allocator aware.

- Like `optional` and `unique_ptr`, `polymorphic` can be in a valueless state; `polymorphic` can get into the valueless state only after being moved from, or after assignment or construction from a valueless state.
- `unique_ptr` and `optional` have preconditions for `operator->` and `operator*`: the behavior is undefined if `*this` does not contain a value.
- `unique_ptr` and `optional` mark `operator->` and `operator*` as noexcept: `polymorphic` does the same.
- Neither `unique_ptr` nor `polymorphic` know the underlying type of the owned object so cannot implement r-value qualified versions of `operator*`. For `optional`, the underlying type is known, so r-value qualified versions of `operator*` are provided.
- Like `vector`, `polymorphic` owns an object created by an allocator. The move constructor and move assignment operator for `vector` are conditionally noexcept on properties of the allocator. Thus for `polymorphic`, the move constructor and move assignment operator are conditionally noexcept on properties of the allocator. Like `vector`, `polymorphic` marks member and non-member `swap` as noexcept and requires allocators to be equal.
- Like `unique_ptr`, `polymorphic` does not know the type of the owned object (it could be an instance of a derived type). As a result, `polymorphic` cannot forward comparison operators or hash to the owned object.

Similarities and differences with variant

The sum type `variant<Ts...>` models one of several alternatives; `indirect<T>` models a single type `T`, but with different storage constraints to `T`.

Like `indirect`, a variant can get into a valueless state. For `variant`, this valueless state is accessible when an exception is thrown when changing the type: variant has `bool valueless_by_exception()`. When all of the types `Ts` are comparable, `variant<Ts...>` supports comparison without preconditions: it is valid to compare variants when they are in a valueless state. Variant comparisons can account for the valueless state with zero cost. A variant must check which type is the engaged type to perform comparison; valueless is one of the possible states it can be in. For `indirect`, allowing comparison when in a valueless state necessitates the addition of an otherwise redundant check. After feedback from standard library implementers, we opt to allow hash and comparison of `indirect` in a valueless state, at cost, to avoid making comparison or hash of `indirect` in a valueless state undefined behaviour.

`variant` allows valueless objects to be passed around via copy, assignment, move and move assignment. There is no precondition on `variant` that it must not be in a valueless state to be copied from, moved from, assigned from or move assigned from. While the notion that a valueless `indirect` or `polymorphic` is toxic and must not be passed around code is appealing, it would not interact well with generic code which may need to handle a variety of types. Note that the standard does not require a moved-from object to be valid for copy, move, assign or move assignment: the restriction is only that it should be in a well-formed but unspecified state. However, there is no precedent for standard library types to have preconditions on move, copy, assign or move assignment. We opt for consistency with existing standard library types (namely `variant`, which has a valueless state) and allow copy, move, assignment and move assignment of a valueless `indirect` and `polymorphic`. Handling of the valueless state for `indirect` and `polymorphic` in move operations will not incur cost; for copy operations, the cost of handling the valueless state will be insignificant compared to the cost of allocating memory. Introducing preconditions for copy, move, assign and move assign in a later revision of the C++ standard would be a silent breaking change.

Like `variant`, `indirect` does not support formatting by forwarding to the owned object. There may be no owned object to format so we require the user to write code to determine how to format a valueless `indirect` or to validate that the `indirect` is not valueless before formatting `*i` (where `i` is an instance of `indirect` for some formattable type `T`).

`noexcept` and narrow contracts

C++ library design guidelines recommend that member functions with narrow contracts (runtime preconditions) should not be marked `noexcept`. This is partially motivated by a non-vendor implementation of the C++ standard library that uses exceptions in a debug build to check for precondition violations by throwing an exception. The `noexcept` status of `operator->` and `operator*` for `indirect` and `polymorphic` is identical to that of `optional` and `unique_ptr`. All have preconditions (`*this` cannot be valueless), all are marked `noexcept`. Whatever strategy was used for testing `optional` and `unique_ptr` can be used for `indirect` and `polymorphic`.

Not marking `operator->` and `operator*` as `noexcept` for `indirect` and `polymorphic` would make them strictly less useful than `unique_ptr` in contexts where they would otherwise be a valid replacement.

Tagged constructors

Constructors for `indirect` and `polymorphic` taking an allocator or owned-object constructor arguments are tagged with `allocator_arg_t` and `in_place_t` (or `in_place_type_t`) respectively. This is consistent with the standard library's use of tagged constructors in `optional`, `any` and `variant`.

Without `in_place_t` the constructor of `indirect` would not be able to construct an owned object using the owned object's allocator-extended constructor. `indirect(std::in_place, std::allocator_arg, alloc, args)` constructs an `indirect` with a default-constructed allocator and an owned object constructed with an allocator-extended constructor taking an allocator `alloc` and constructor arguments `args`.

For comparison, `indirect(std::allocator_arg, a, std::in_place, std::allocator_arg, alloc, args)` constructs an `indirect` with an allocator `a` and an owned object constructed with an allocator-extended constructor taking an allocator `alloc` and constructor arguments `args`.

Single-argument constructors

In line with `optional` and `variant`, we add single-argument constructors to both `indirect` and `polymorphic` so they can be constructed from single values without the need to use `in_place` or `in_place_type`. As `indirect` and `polymorphic` are allocator-aware types, we also provide allocator-extended versions of these constructors, in line with those from `basic_optional` [2] and existing constructors from `indirect` and `polymorphic`.

Initializer-list constructors

We add initializer-list constructors to both `indirect` and `polymorphic` in line with those in `optional` and `variant`. As `indirect` and `polymorphic` are allocator-aware types, we provide allocator-extended versions of these constructors, in line with those from `basic_optional` [2] and existing constructors from `indirect` and `polymorphic`.

Explicit constructors

Constructors for `indirect` and `polymorphic` are marked as explicit. This disallows "implicit conversion" from single arguments or braced initializers. Given both `indirect` and `polymorphic` use dynamically-allocated storage, there are no instances where an object could be considered semantically equivalent to its constructor arguments (unlike `pair` or `variant`). To construct an `indirect` or `polymorphic` object, and with it use dynamically-allocated memory, the user must explicitly use a constructor.

The standard already marks multiple argument constructors as explicit for the inplace constructors of `optional` and `any`.

With some suitably compelling motivation, the `explicit` keyword could be removed from some constructors in a later revision of the C++ standard without rendering code ill-formed.

Perfect-forwarded assignment

Perfect-forwarded assignment for `indirect`

We add a perfect-forwarded assignment operator for `indirect` in line with those from `optional` and `variant`.

```
template <class U=T>
constexpr optional& operator=(U&& u);
```

When assigning to an `indirect`, there is potential for optimisation if there is an existing owned object to be assigned to:

```
indirect<int> i;
foo(i);  // could move from `i`.
if (!i.valueless_after_move()) {
  *i = 5;
} else {
  i = indirect(5);
}
```

With perfect-forwarded assignment, handling the valueless state and potentially creating a new indirect object is done within the perfect-forwarded assignment. The code below is equivalent to the code above:

```
indirect<int> i;
foo(i); // could move from `i`.
i = 5;
```

Perfect-forwarded assignment for `polymorphic`

There is no perfect-forwarded assignment for `polymorphic` as type information is erased. There is no optimisation opportunity to be made as a new object will need creating regardless of whether the target of assignment is valueless or not.

The `valueless_after_move` member function

Both `indirect` and `polymorphic` have a `valueless_after_move` member function that is used to query the object state. This member function should rarely be called: it should be clear through static analysis whether or not an object has been moved from. The `valueless_after_move` member function allows explicit checks for the valueless state in cases where it cannot be verified statically or where explicit checks might be required by a coding standard such as MISRA or High Integrity C++.

Design for polymorphic types

A type `PolymorphicInterface` used as a base class with `polymorphic` does not need a virtual destructor. The same mechanism that is used to call the copy constructor of a potentially derived-type object will be used to call the destructor.

To allow compiler-generation of special member functions of an abstract interface type `PolymorphicInterface` in conjunction with `polymorphic`, `PolymorphicInterface` needs at least a non-virtual protected destructor and a protected copy constructor. `PolymorphicInterface` does not need to be assignable, move constructible or move assignable for `polymorphic<PolymorphicInterface>` to be assignable, move constructible or move assignable.

```
class PolymorphicInterface {
  protected:
    PolymorphicInterface(const PolymorphicInterface&) = default;
    ~PolymorphicInterface() = default;
  public:
    // virtual functions
};
```

For an interface type with a public virtual destructor, users would potentially pay the cost of virtual dispatch twice when deleting `polymorphic<I>` objects containing derived-type objects.

All derived types owned by a `polymorphic` must be publicly copy constructible.

Prior work

This proposal continues the work started in [P0201] and [P1950].

Previous work on a cloned pointer type [N3339] met with opposition because of the mixing of value and pointer semantics. We believe that the unambiguous value semantics of `indirect` and `polymorphic` as described in this proposal address these concerns.

Impact on the standard

This proposal is a pure library extension. It requires additions to be made to the standard library header `<memory>`.

Technical specifications

Header `<version>` synopsis [version.syn]

Note to editors: Add the following macros with editor provided values to [version.syn]

```
#define __cpp_lib_indirect ??????L    // also in <memory>
#define __cpp_lib_polymorphic ??????L // also in <memory>
```

Header `<memory>` synopsis [memory]

```
namespace std {

  // [inout.ptr], function template inout_ptr
  template<class Pointer = void, class Smart, class... Args>
    auto inout_ptr(Smart& s, Args&&... args);

<ins>
// DRAFTING NOTE: not sure how to typeset <ins> reasonably in markdown
  // [indirect], class template indirect
  template<class T, class Allocator = allocator<T>>
    class indirect;

  // [indirect.hash], hash support
  template <class T, class Alloc> struct hash<indirect<T, Alloc>>;

  // [polymorphic], class template polymorphic
  template <class T, class Allocator = allocator<T>>
    class polymorphic;
  namespace pmr {

    template<class T> using indirect =
      indirect<T, polymorphic_allocator<T>>;

    template<class T> using polymorphic =
      polymorphic<T, polymorphic_allocator<T>>;
  }
</ins>
}
```

X.Y Class template indirect [indirect]

[Drafting note: The member *alloc* should be formatted as an exposition only identifier, but limitations of the processor used to prepare this paper means not all uses are italicised.]

X.Y.1 Class template indirect general [indirect.general]

1. An indirect object manages the lifetime of an owned object. An indirect object is *valueless* if it has no owned object. An indirect object may become valueless only after it has been moved from.

2. In every specialization `indirect<T, Allocator>`, if the type `allocator_traits<Allocator>::value_type` is not the same type as `T`, the program is ill-formed. Every object of type `indirect<T, Allocator>` uses an object of type `Allocator` to allocate and free storage for the owned object as needed.

3. Constructing an owned object with `args...` using the allocator `a` means calling `allocator_traits<Allocator>::construct(a, p, args...)` where `args` is an expression pack, `a` is an allocator, and `p` is a pointer obtained by calling `allocator_traits<Allocator>::allocate`.

4. The member `alloc` is used for any memory allocation and element construction performed by member functions during the lifetime of each indirect object. The allocator `alloc` may be replaced only via assignment or `swap()`. Allocator replacement is performed by copy assignment, move assignment, or swapping of the allocator only if ([container.reqmts]): `allocator_traits<Allocator>::propagate_on_container_copy_assignment::value`, or `allocator_traits<Allocator>::propagate_on_container_move_assignment::value`, or `allocator_traits<Allocator>::propagate_on_container_swap::value` is `true` within the implementation of the corresponding indirect operation.

5. A program that instantiates the definition of the template `indirect<T, Allocator>` with a type for the `T` parameter that is a non-object type, an array type, `in_place_t`, a specialization of `in_place_type_t`, or a cv-qualified type is ill-formed.

6. The template parameter `T` of `indirect` may be an incomplete type.

7. The template parameter `Allocator` of `indirect` shall meet the *Cpp17Allocator* requirements.

8. If a program declares an explicit or partial specialization of `indirect`, the behavior is undefined.

X.Y.2 Class template indirect synopsis [indirect.syn]

```
template <class T, class Allocator = allocator<T>>
class indirect {
 public:
  using value_type = T;
  using allocator_type = Allocator;
  using pointer = typename allocator_traits<Allocator>::pointer;
  using const_pointer = typename allocator_traits<Allocator>::const_pointer;

  explicit constexpr indirect();

  explicit constexpr indirect(allocator_arg_t, const Allocator& a);
```

```
constexpr indirect(const indirect& other);

constexpr indirect(allocator_arg_t, const Allocator& a,
                   const indirect& other);

constexpr indirect(indirect&& other) noexcept;

constexpr indirect(allocator_arg_t, const Allocator& a,
                   indirect&& other) noexcept(see below);

template <class U=T>
explicit constexpr indirect(U&& u);

template <class U=T>
explicit constexpr indirect(allocator_arg_t, const Allocator& a, U&& u);

template <class... Us>
explicit constexpr indirect(in_place_t, Us&&... us);

template <class... Us>
explicit constexpr indirect(allocator_arg_t, const Allocator& a,
                            in_place_t, Us&&... us);

template<class I, class... Us>
explicit constexpr indirect(in_place_t, initializer_list<I> ilist,
                            Us&&... us);

template<class I, class... Us>
explicit constexpr indirect(allocator_arg_t, const Allocator& a,
                            in_place_t, initializer_list<I> ilist,
                            Us&&... us);

constexpr ~indirect();

constexpr indirect& operator=(const indirect& other);

constexpr indirect& operator=(indirect&& other) noexcept(see below);

template <class U=T>
constexpr indirect& operator=(U&& u);

constexpr const T& operator*() const & noexcept;

constexpr T& operator*() & noexcept;

constexpr const T&& operator*() const && noexcept;

constexpr T&& operator*() && noexcept;

constexpr const_pointer operator->() const noexcept;

constexpr pointer operator->() noexcept;

constexpr bool valueless_after_move() const noexcept;
```

```
  constexpr allocator_type get_allocator() const noexcept;

  constexpr void swap(indirect& other) noexcept(see below);

  friend constexpr void swap(indirect& lhs, indirect& rhs) noexcept(see below);

  template <class U, class AA>
  friend constexpr bool operator==(
    const indirect& lhs, const indirect<U, AA>& rhs) noexcept(see below);

  template <class U>
  friend constexpr bool operator==(
    const indirect& lhs, const U& rhs) noexcept(see below);

  template <class U, class AA>
  friend constexpr auto operator<=>(
    const indirect& lhs, const indirect<U, AA>& rhs)
    -> synth-three-way-result<T, U>;

  template <class U>
  friend constexpr auto operator<=>(
    const indirect& lhs, const U& rhs)
    -> synth-three-way-result<T, U>;

private:
  pointer p; // exposition only
  Allocator alloc = Allocator(); // exposition only
};

template <class Value>
indirect(Value) -> indirect<Value>;

template <class Allocator, class Value>
indirect(allocator_arg_t, Allocator, Value) -> indirect<Value,
         typename allocator_traits<Allocator>::template rebind_alloc<Value>>;
```

X.Y.3 Constructors [indirect.ctor]

The following element applies to all functions in [indirect.ctor]:

Throws: Nothing unless `allocator_traits<Allocator>::allocate` or `allocator_traits<Allocator>::construct` throws.

```
explicit constexpr indirect();
```

1. *Constraints*: `is_default_constructible_v<Allocator>` is `true`.
2. *Mandates*: `is_default_constructible_v<T>` is `true`.
3. *Effects*: Constructs an owned object of type `T` with an empty argument list, using the allocator `alloc`.

```
explicit constexpr indirect(allocator_arg_t, const Allocator& a);
```

4. *Mandates*: `is_default_constructible_v<T>` is `true`.
5. *Effects*: `alloc` is direct-non-list-initialized with `a`. Constructs an owned object of type `T` with an empty argument list, using the allocator `alloc`.

```
constexpr indirect(const indirect& other);
```

6. *Mandates*: `is_copy_constructible_v<T>` is `true`.

7. *Effects*: `alloc` is direct-non-list-initialized with `allocator_traits<Allocator>::select_on_container_copy_construction(other.alloc)`. If `other` is valueless, `*this` is valueless. Otherwise, constructs an owned object of type `T` with `*other`, using the allocator `alloc`.

```
constexpr indirect(allocator_arg_t, const Allocator& a,
                   const indirect& other);
```

8. *Mandates*: `is_copy_constructible_v<T>` is `true`.

9. *Effects*: `alloc` is direct-non-list-initialized with `a`. If `other` is valueless, `*this` is valueless. Otherwise, constructs an owned object of type `T` with `*other`, using the allocator `alloc`.

```
constexpr indirect(indirect&& other) noexcept;
```

10. *Effects*: `alloc` is direct-non-list-initialized from `std::move(other.alloc)`. If `other` is valueless, `*this` is valueless. Otherwise `*this` takes ownership of the owned object of `other`.

11. *Postconditions*: `other` is valueless.

```
constexpr indirect(allocator_arg_t, const Allocator& a, indirect&& other)
  noexcept(allocator_traits<Allocator>::is_always_equal::value);
```

12. *Mandates*: If `allocator_traits<Allocator>::is_always_equal::value` is `false` then `T` is a complete type.

13. *Effects*: `alloc` is direct-non-list-initialized with `a`. If `other` is valueless, `*this` is valueless. Otherwise, if `alloc == other.alloc` is `true`, constructs an object of type `indirect` that takes ownership of the owned object of `other`. Otherwise, constructs an owned object of type `T` with `*std::move(other)`, using the allocator `alloc`.

14. *Postconditions*: `other` is valueless.

```
template <class U=T>
explicit constexpr indirect(U&& u);
```

15. *Constraints*:
    - `is_same_v<remove_cvref_t<U>, indirect>` is `false`,
    - `is_same_v<remove_cvref_t<U>, in_place_t>` is `false`,
    - `is_constructible_v<T, U>` is `true`, and
    - `is_default_constructible_v<Allocator>` is `true`.
16. *Effects*: Constructs an owned object of type `T` with `std::forward<U>(u)`, using the allocator `alloc`.

```
template <class U=T>
explicit constexpr indirect(allocator_arg_t, const Allocator& a, U&& u);
```

17. *Constraints*:
    - `is_same_v<remove_cvref_t<U>, indirect>` is `false`,
    - `is_same_v<remove_cvref_t<U>, in_place_t>` is `false`, and
    - `is_constructible_v<T, U>` is `true`.
18. *Effects*: `alloc` is direct-non-list-initialized with `a`. Constructs an owned object of type `T` with `std::forward<U>(u)`, using the allocator `alloc`.

```
template <class... Us>
explicit constexpr indirect(in_place_t, Us&&... us);
```

19. *Constraints*:
    - `is_constructible_v<T, Us...>` is `true`, and
    - `is_default_constructible_v<Allocator>` is `true`.
20. *Effects*: Constructs an owned object of type `T` with `std::forward<Us>(us)...`, using the allocator `alloc`.

```
template <class... Us>
explicit constexpr indirect(allocator_arg_t, const Allocator& a,
                            in_place_t, Us&& ...us);
```

21. *Constraints*: `is_constructible_v<T, Us...>` is `true`.

22. *Effects*: `alloc` is direct-non-list-initialized with `a`. Constructs an owned object of type `T` with `std::forward<Us>(us)...`, using the allocator `alloc`.

```
template<class I, class... Us>
explicit constexpr indirect(in_place_t, initializer_list<I> ilist,
                            Us&&... us);
```

23. *Constraints*:
    - `is_constructible_v<T, initializer_list<I>&, Us...>` is `true`, and
    - `is_default_constructible_v<Allocator>` is `true`.
24. *Effects*: Constructs an owned object of type `T` with the arguments `ilist`, `std::forward<Us>(us)...`, using the allocator `alloc`.

```
template<class I, class... Us>
explicit constexpr indirect(allocator_arg_t, const Allocator& a,
                            in_place_t, initializer_list<I> ilist,
                            Us&&... us);
```

25. *Constraints*: `is_constructible_v<T, initializer_list<I>&, Us...>` is `true`.

26. *Effects*: `alloc` is direct-non-list-initialized with `a`. Constructs an owned object of type `T` with the arguments `ilist`, `std::forward<Us>(us)...`, using the allocator `alloc`.

X.Y.4 Destructor [indirect.dtor]

```
constexpr ~indirect();
```

1. *Mandates*: `T` is a complete type.

2. *Effects*: If `*this` is not valueless, destroys the owned object using `allocator_traits<Allocator>::destroy` and then the storage is deallocated.

X.Y.5 Assignment [indirect.assign]

```
constexpr indirect& operator=(const indirect& other);
```

1. *Mandates*:

    - `is_copy_assignable_v<T>` is `true`, and
    - `is_copy_constructible_v<T>` is `true`.

2. *Effects*: If `addressof(other) == this` is `true`, there are no effects.
   Otherwise:

   2.1. The allocator needs updating if
   `allocator_traits<Allocator>::propagate_on_container_copy_assignment::value`
   is `true`.

   2.2. If `other` is valueless, `*this` becomes valueless and the owned object in `*this`, if any, is destroyed using `allocator_traits<Allocator>::destroy` and then the storage is deallocated.

   2.3. Otherwise, if `alloc == other.alloc` is `true` and `*this` is not valueless, equivalent to `**this = *other`.

   2.4. Otherwise a new owned object is constructed in `*this` using `allocator_traits<Allocator>::construct` with the owned object from `other` as the argument, using either the allocator in `*this` or the allocator in `other` if the allocator needs updating.

   2.5. The previously owned object in `*this`, if any, is destroyed using `allocator_traits<Allocator>::destroy` and then the storage is deallocated.

   2.6. If the allocator needs updating, the allocator in `*this` is replaced with a copy of the allocator in `other`.

3. *Returns*: A reference to `*this`.

4. *Remarks*: If any exception is thrown, the result of the expression `this->valueless_after_move()` remains unchanged. If an exception is thrown during the call to `T`'s selected copy constructor, no effect. If an exception

is thrown during the call to T's copy assignment, the state of its contained value is as defined by the exception safety guarantee of T's copy assignment.

```
constexpr indirect& operator=(indirect&& other) noexcept(
    allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
    allocator_traits<Allocator>::is_always_equal::value);
```

5. *Mandates*: `is_copy_constructible_t<T>` is `true`.
6. *Effects*: If `addressof(other) == this` is `true`, there are no effects. Otherwise:

   6.1. The allocator needs updating if
   `allocator_traits<Allocator>::propagate_on_container_move_assignment::value`
   is `true`.

   6.2. If `other` is valueless, `*this` becomes valueless and the owned object in `*this`, if any, is destroyed using `allocator_traits<Allocator>::destroy` and then the storage is deallocated.

   6.3. Otherwise, if `alloc == other.alloc` is `true`, swaps the owned objects in `*this` and `other`; the owned object in `other`, if any, is then destroyed using `allocator_traits<Allocator>::destroy` and then the storage is deallocated.

   6.4. Otherwise constructs a new owned object with the owned object of `other` as the argument as an rvalue, using either the allocator in `*this` or the allocator in `other` if the allocator needs updating.

   6.5. The previously owned object in `*this`, if any, is destroyed using `allocator_traits<Allocator>::destroy` and then the storage is deallocated.

   6.6. If the allocator needs updating, the allocator in `*this` is replaced with a copy of the allocator in `other`.
7. *Postconditions*: `other` is valueless.
8. *Returns*: A reference to `*this`.
9. *Remarks*: If any exception is thrown, there are no effects on `*this` or `other`.

```
template <class U=T>
constexpr indirect& operator=(U&& u);
```

10. *Constraints*:
    - `is_same_v<remove_cvref_t<U>, indirect>` is `false`,
    - `is_constructible_v<T, U>` is `true`, and
    - `is_assignable_v<T&, U>` is `true`.
11. *Effects*: If `*this` is valueless then constructs an owned object of type `T` with `std::forward<U>(u)` using the allocator `alloc`. Otherwise, equivalent to
    `**this = std::forward<U>(u)`.
12. *Returns*: A reference to `*this`.

X.Y.6 Observers [indirect.observers]

```
constexpr const T& operator*() const & noexcept;
constexpr T& operator*() & noexcept;
```

1. *Preconditions*: `*this` is not valueless.
2. *Returns*: `*p`.

```
constexpr const T&& operator*() const && noexcept;
constexpr T&& operator*() && noexcept;
```

3. *Preconditions*: `*this` is not valueless.
4. *Returns*: `std::move(*p)`.

```
constexpr const_pointer operator->() const noexcept;
constexpr pointer operator->() noexcept;
```

5. *Preconditions*: `*this` is not valueless.

6. *Returns*: `p`.

```
constexpr bool valueless_after_move() const noexcept;
```

7. *Returns*: `true` if `*this` is valueless, otherwise `false`.

```
constexpr allocator_type get_allocator() const noexcept;
```

8. *Returns*: `alloc`.

X.Y.7 Swap [indirect.swap]

```
constexpr void swap(indirect& other) noexcept(
  allocator_traits<Allocator>::propagate_on_container_swap::value
  || allocator_traits<Allocator>::is_always_equal::value);
```

1. *Preconditions*: If
   `allocator_traits<Allocator>::propagate_on_container_swap::value`
   is `true`, then `Allocator` meets the *Cpp17Swappable* requirements. Otherwise `get_allocator() == other.get_allocator()` is `true`.

2. *Effects*: Swaps the states of `*this` and `other`, exchanging owned objects or valueless states. If
   `allocator_traits<Allocator>::propagate_on_container_swap::value`
   is `true`, then the allocators of `*this` and `other` are exchanged by calling `swap` as described in [swappable.requirements]. Otherwise, the allocators are not swapped. *[Note: Does not call* `swap` *on the owned objects directly. –end note]*

```
constexpr void swap(indirect& lhs, indirect& rhs) noexcept(
  noexcept(lhs.swap(rhs)));
```

3. *Effects*: Equivalent to `lhs.swap(rhs)`.

X.Y.8 Relational operators [indirect.relops]

```
template <class U, class AA>
constexpr bool operator==(const indirect& lhs, const indirect<U, AA>& rhs)
  noexcept(noexcept(*lhs == *rhs));
```

1. *Mandates*: The expression `*lhs == *rhs` is well-formed and its result is convertible to bool.

2. *Returns*: If `lhs` is valueless or `rhs` is valueless,
   `lhs.valueless_after_move() == rhs.valueless_after_move()`; otherwise `*lhs == *rhs`.

```
template <class U, class AA>
constexpr synth-three-way-result<T, U> operator<=>(const indirect& lhs,
                                                   const indirect<U, AA>& rhs);
```

3. *Returns*: If `lhs` is valueless or `rhs` is valueless,
   `!lhs.valueless_after_move() <=> !rhs.valueless_after_move()`; otherwise
   `synth-three-way(*lhs, *rhs)`.

X.Y.9 Comparison with T [indirect.comp.with.t]

```
template <class U>
constexpr bool operator==(const indirect& lhs, const U& rhs)
  noexcept(noexcept(*lhs == rhs));
```

1. *Mandates*: The expression `*lhs == rhs` is well-formed and its result is convertible to bool.

2. *Returns*: If `lhs` is valueless, false; otherwise `*lhs == rhs`.

```
template <class U>
constexpr synth-three-way-result<T, U> operator<=>(const indirect& lhs,
                                                   const U& rhs);
```

3. *Returns*: If `lhs` is valueless, `strong_ordering::less`; otherwise `synth-three-way(*lhs, rhs)`.

X.Y.10 Hash support [indirect.hash]

```
template <class T, class Allocator>
struct hash<indirect<T, Allocator>>;
```

1. The specialization `hash<indirect<T, Allocator>>` is enabled ([unord.hash]) if and only if `hash<T>` is enabled. When enabled for an object `i` of type `indirect<T, Allocator>`, then `hash<indirect<T, Allocator>>()(i)` evaluates to either the same value as `hash<T>()(*i)`, if `i` is not valueless; otherwise to an implementation-defined value. The member functions are not guaranteed to be noexcept.

X.Z Class template polymorphic [polymorphic]

[Drafting note: The member *`alloc`* should be formatted as an exposition only identifier, but limitations of the processor used to prepare this paper mean not all uses are italicised.]

X.Z.1 Class template polymorphic general [polymorphic.general]

1. A polymorphic object manages the lifetime of an owned object. A polymorphic object may own objects of different types at different points in its lifetime. A polymorphic object is *valueless* if it has no owned object. A polymorphic object may become valueless only after it has been moved from.

2. In every specialization `polymorphic<T, Allocator>`, if the type `allocator_traits<Allocator>::value_type` is not the same type as`T`, the program is ill-formed. Every object of type `polymorphic<T, Allocator>` uses an object of type `Allocator` to allocate and free storage for the owned object as needed.

3. Constructing an owned object of type `U` with `args...` using the allocator `a` means calling `allocator_traits<Allocator>::co` `p, args...)` where `args` is an expression pack, `a` is an allocator, `p` points to storage suitable for an owned object of type `U`.

4. The member `alloc` is used for any memory allocation and element construction performed by member functions during the lifetime of each polymorphic value object, or until the allocator is replaced. The allocator may be replaced only via assignment or `swap()`. Allocator replacement is performed by copy assignment, move assignment, or swapping of the allocator only if (see [container.reqmts]):
`allocator_traits<Allocator>::propagate_on_container_copy_assignment::value`,
or
`allocator_traits<Allocator>::propagate_on_container_move_assignment::value`,
or
`allocator_traits<Allocator>::propagate_on_container_swap::value` is true within the implementation of the corresponding polymorphic operation.

5. A program that instantiates the definition of polymorphic for a non-object type, an array type, `in_place_t`, a specialization of `in_place_type_t`, or a cv-qualified type is ill-formed.

6. The template parameter `T` of `polymorphic` may be an incomplete type.

7. The template parameter `Allocator` of `polymorphic` shall meet the requirements of *Cpp17Allocator*.

8. If a program declares an explicit or partial specialization of `polymorphic`, the behavior is undefined.

X.Z.2 Class template polymorphic synopsis [polymorphic.syn]

```
template <class T, class Allocator = allocator<T>>
class polymorphic {
 public:
  using value_type = T;
  using allocator_type = Allocator;
  using pointer = typename allocator_traits<Allocator>::pointer;
  using const_pointer  = typename allocator_traits<Allocator>::const_pointer;

  explicit constexpr polymorphic();
  explicit constexpr polymorphic(allocator_arg_t, const Allocator& a);

  constexpr polymorphic(const polymorphic& other);

  constexpr polymorphic(allocator_arg_t, const Allocator& a,
                        const polymorphic& other);

  constexpr polymorphic(polymorphic&& other) noexcept;

  constexpr polymorphic(allocator_arg_t, const Allocator& a,
                        polymorphic&& other) noexcept(see below);

  template <class U=T>
  explicit constexpr polymorphic(U&& u);

  template <class U=T>
  explicit constexpr polymorphic(allocator_arg_t, const Allocator& a,
                                 U&& u);

  template <class U, class... Ts>
  explicit constexpr polymorphic(in_place_type_t<U>, Ts&&... ts);

  template <class U, class... Ts>
  explicit constexpr polymorphic(allocator_arg_t, const Allocator& a,
                                 in_place_type_t<U>, Ts&&... ts);

  template <class U, class I, class... Us>
  explicit constexpr polymorphic(in_place_type_t<U>,
                                 initializer_list<I> ilist, Us&&... us);

  template <class U, class I, class... Us>
  explicit constexpr polymorphic(allocator_arg_t, const Allocator& a,
                                 in_place_type_t<U>,
                                 initializer_list<I> ilist, Us&&... us);

  constexpr ~polymorphic();

  constexpr polymorphic& operator=(const polymorphic& other);

  constexpr polymorphic& operator=(polymorphic&& other) noexcept(see below);

  constexpr const T& operator*() const noexcept;

  constexpr T& operator*() noexcept;

  constexpr const_pointer operator->() const noexcept;

  constexpr pointer operator->() noexcept;

  constexpr bool valueless_after_move() const noexcept;

  constexpr allocator_type get_allocator() const noexcept;

  constexpr void swap(polymorphic& other) noexcept(see below);

  friend constexpr void swap(polymorphic& lhs,
                             polymorphic& rhs) noexcept(see below);
 private:
  Allocator alloc = Allocator(); // exposition only
};
```

X.Z.3 Constructors [polymorphic.ctor]

The following element applies to all functions in [polymorphic.ctor]:

Throws: Nothing unless `allocator_traits<Allocator>::allocate` or `allocator_traits<Allocator>::construct` throws.

```
explicit constexpr polymorphic();
```

1. *Constraints*: `is_default_constructible_v<Allocator>` is `true`.
2. *Mandates*:
   - `is_default_constructible_v<T>` is `true`, and
   - `is_copy_constructible_v<T>` is `true`.
3. *Effects*: Constructs an owned object of type `T` with an empty argument list using the allocator `alloc`.

```
explicit constexpr polymorphic(allocator_arg_t, const Allocator& a);
```

4. *Mandates*:
   - `is_default_constructible_v<T>` is `true`, and
   - `is_copy_constructible_v<T>` is `true`.
5. *Effects*: `alloc` is direct-non-list-initialized with `a`. Constructs an owned object of type `T` with an empty argument list using the allocator `alloc`.

```
constexpr polymorphic(const polymorphic& other);
```

6. *Effects*: `alloc` is direct-non-list-initialized with `allocator_traits<Allocator>::select_on_container_copy_construction(other.alloc)`. If `other` is valueless, `*this` is valueless. Otherwise, constructs an owned object of type `U`, where `U` is the type of the owned object in `other`, with the owned object in `other` using the allocator `alloc`.

```
constexpr polymorphic(allocator_arg_t, const Allocator& a,
                      const polymorphic& other);
```

7. *Effects*: `alloc` is direct-non-list-initialized with `a`. If `other` is valueless, `*this` is valueless. Otherwise, constructs an owned object of type `U`, where `U` is the type of the owned object in `other`, with the owned object in `other` using the allocator `alloc`.

```
constexpr polymorphic(polymorphic&& other) noexcept;
```

8. *Effects*: `alloc` is direct-non-list-initialized with `std::move(other.alloc)`. If `other` is valueless, `*this` is valueless. Otherwise, either `*this` takes ownership of the owned object of `other` or, owns an object of the same type constructed from the owned object of `other` considering that owned object as an rvalue, using the allocator `alloc`.

[Drafting note: The above is intended to permit a small-buffer-optimization and handle the case where allocators compare equal but we do not want to swap pointers.]

```
constexpr polymorphic(allocator_arg_t, const Allocator& a,
                      polymorphic&& other)
  noexcept(allocator_traits<Allocator>::is_always_equal::value);
```

9. *Effects*: `alloc` is direct-non-list-initialized with `a`. If `other` is valueless, `*this` is valueless. Otherwise, if `alloc == other.alloc` is `true`, either constructs an object of type `polymorphic` that owns the owned object of other, making `other` valueless; or, owns an object of the same type constructed from the owned object of `other` considering that owned object as an rvalue. Otherwise, if `alloc != other.alloc` is `true`, constructs an object of type `polymorphic`, considering the owned object in `other` as an rvalue, using the allocator `alloc`.

[Drafting note: The above is intended to permit a small-buffer-optimization and handle the case where allocators compare equal but we do not want to swap pointers.]

```
template <class U=T>
explicit constexpr polymorphic(U&& u);
```

10. *Constraints*: Where `UU` is `remove_cvref_t<U>`,
    - `is_same_v<UU, polymorphic>` is `false`,
    - `derived_from<UU, T>` is `true`,
    - `is_constructible_v<UU, U>` is `true`,
    - `is_copy_constructible_v<UU>` is `true`,
    - `UU` is not a specialization of `in_place_type_t`, and
    - `is_default_constructible_v<Allocator>` is `true`.
11. *Effects*: Constructs an owned object of type `U` with `std::forward<U>(u)` using the allocator `alloc`.

```
template <class U=T>
explicit constexpr polymorphic(allocator_arg_t, const Allocator& a, U&& u);
```

12. *Constraints*: Where `UU` is `remove_cvref_t<U>`,
    - `is_same_v<UU, polymorphic>` is `false`,
    - `derived_from<UU, T>` is `true`,
    - `is_constructible_v<UU, U>` is `true`,
    - `is_copy_constructible_v<UU>` is `true`, and
    - `UU` is not a specialization of `in_place_type_t`.
13. *Effects*: `alloc` is direct-non-list-initialized with `a`. Constructs an owned object of type `U` with `std::forward<U>(u)` using the allocator `alloc`.

```
template <class U, class... Ts>
explicit constexpr polymorphic(in_place_type_t<U>, Ts&&... ts);
```

14. *Constraints*:
    - `is_same_v<remove_cvref_t<U>, U>` is `true`,
    - `derived_from<U, T>` is `true`,
    - `is_constructible_v<U, Ts...>` is `true`,
    - `is_copy_constructible_v<U>` is `true`, and
    - `is_default_constructible_v<Allocator>` is `true`.
15. *Effects*: Constructs an owned object of type `U` with `std::forward<Ts>(ts)...` using the allocator `alloc`.

```
template <class U, class... Ts>
explicit constexpr polymorphic(allocator_arg_t, const Allocator& a,
                               in_place_type_t<U>, Ts&&... ts);
```

16. *Constraints*:
    - `is_same_v<remove_cvref_t<U>, U>` is `true`,
    - `derived_from<U, T>` is `true`,
    - `is_constructible_v<U, Ts...>` is `true`, and
    - `is_copy_constructible_v<U>` is `true`.
17. *Effects*: `alloc` is direct-non-list-initialized with `a`. Constructs an owned object of type `U` with `std::forward<Ts>(ts)...` using the allocator `alloc`.

```
template <class U, class I, class... Us>
explicit constexpr polymorphic(in_place_type_t<U>,
                               initializer_list<I> ilist, Us&&... us);
```

18. *Constraints*:
    - `is_same_v<remove_cvref_t<U>, U>` is `true`,
    - `derived_from<U, T>` is `true`,
    - `is_constructible_v<U, initializer_list<I>&, Us...>` is `true`,
    - `is_copy_constructible_v<U>` is `true`, and
    - `is_default_constructible_v<Allocator>` is `true`.
19. *Effects*: Constructs an owned object of type `U` with the arguments `ilist`, `std::forward<Us>(us)...` using the allocator `alloc`.

```
template <class U, class I, class... Us>
explicit constexpr polymorphic(allocator_arg_t, const Allocator& a,
                               in_place_type_t<U>,
                               initializer_list<I> ilist, Us&&... us);
```

20. *Constraints*:
    - `is_same_v<remove_cvref_t<U>, U>` is `true`,
    - `derived_from<U, T>` is `true`,
    - `is_constructible_v<U, initializer_list<I>&, Us...>` is `true`, and
    - `is_copy_constructible_v<U>` is `true`.
21. *Effects*: `alloc` is direct-non-list-initialized with `a`. Constructs an owned object of type `U` with the arguments `ilist, std::forward<Us>(us)...` using the allocator `alloc`.

X.Z.4 Destructor [polymorphic.dtor]

```
constexpr ~polymorphic();
```

1. *Mandates*: `T` is a complete type.
2. *Effects*: If `*this` is not valueless, destroys the owned object using `allocator_traits<Allocator>::destroy` and then the storage is deallocated.

X.Z.5 Assignment [polymorphic.assign]

```
constexpr polymorphic& operator=(const polymorphic& other);
```

1. *Mandates*: `T` is a complete type.
2. *Effects*: If `addressof(other) == this` is `true`, there are no effects. Otherwise:

   2.1. The allocator needs updating if
   `allocator_traits<Allocator>::propagate_on_container_copy_assignment::value`
   is `true`.

   2.2. If `other` is not valueless, a new owned object is constructed in `*this` using `allocator_traits<Allocator>::construct` with the owned object from `other` as the argument, using either the allocator in `*this` or the allocator in `other` if the allocator needs updating.

   2.3 The previously owned object in `*this`, if any, is destroyed using `allocator_traits<Allocator>::destroy` and then the storage is deallocated.

   2.4 If the allocator needs updating, the allocator in `*this` is replaced with a copy of the allocator in `other`.
3. *Returns*: A reference to `*this`.
4. *Remarks*: If any exception is thrown, there are no effects on `*this`.

```
constexpr polymorphic& operator=(polymorphic&& other) noexcept(
    allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
    allocator_traits<Allocator>::is_always_equal::value);
```

5. *Mandates*: If `allocator_traits<Allocator>::is_always_equal::value` is `false`, `T` is a complete type.
6. *Effects*: If `addressof(other) == this` is `true`, there are no effects. Otherwise:

   6.1. The allocator needs updating if
   `allocator_traits<Allocator>::propagate_on_container_move_assignment::value`
   is `true`.

   6.2. If `alloc == other.alloc` is `true`, swaps the owned objects in `*this` and `other`; the owned object in `other`, if any, is then destroyed using `allocator_traits<Allocator>::destroy` and then the storage is deallocated.

   6.3. Otherwise, if `alloc != other.alloc` is `true`; if `other` is not valueless, a new owned object is constructed in `*this` using `allocator_traits<Allocator>::construct` with the owned object from `other` as the argument as an rvalue, using either the allocator in `*this` or the allocator in `other` if the allocator needs updating.

6.4. The previously owned object in `*this`, if any, is destroyed using `allocator_traits<Allocator>::destroy` and then the storage is deallocated.

6.5. If the allocator needs updating, the allocator in `*this` is replaced with a copy of the allocator in `other`.

7. *Returns*: A reference to `*this`.

8. *Remarks*: If any exception is thrown, there are no effects on `*this` or `other`.

X.Z.6 Observers [polymorphic.observers]

```
constexpr const T& operator*() const noexcept;
constexpr T& operator*() noexcept;
```

1. *Preconditions*: `*this` is not valueless.

2. *Returns*: A reference to the owned object.

```
constexpr const_pointer operator->() const noexcept;
constexpr pointer operator->() noexcept;
```

3. *Preconditions*: `*this` is not valueless.

4. *Returns*: A pointer to the owned object.

```
constexpr bool valueless_after_move() const noexcept;
```

5. *Returns*: `true` if `*this` is valueless, otherwise `false`.

```
constexpr allocator_type get_allocator() const noexcept;
```

6. *Returns*: `alloc`.

X.Z.7 Swap [polymorphic.swap]

```
constexpr void swap(polymorphic& other) noexcept(
  allocator_traits<Allocator>::propagate_on_container_swap::value
  || allocator_traits<Allocator>::is_always_equal::value);
```

1. *Preconditions*: If `allocator_traits<Allocator>::propagate_on_container_swap::value` is `true`, then `Allocator` meets the *Cpp17Swappable* requirements. Otherwise `get_allocator() == other.get_allocator()` is `true`.

2. *Effects*: Swaps the states of `*this` and `other`, exchanging owned objects or valueless states. If `allocator_traits<Allocator>::propagate_on_container_swap::value` is `true`, then the allocators of `*this` and `other` are exchanged by calling `swap` as described in [swappable.requirements]. Otherwise, the allocators are not swapped. *[Note: Does not call `swap` on the owned objects directly. –end note]*

```
constexpr void swap(polymorphic& lhs, polymorphic& rhs) noexcept(
  noexcept(lhs.swap(rhs)));
```

3. *Effects*: Equivalent to `lhs.swap(rhs)`.

Reference implementation

A C++20 reference implementation of this proposal is available on GitHub at https://www.github.com/jbcoe/value_types.

Acknowledgements

The authors would like to thank Andrew Bennieston, Bengt Gustafsson, Casey Carter, Daniel Krugler, David Krauss, David Stone, Ed Catmur, Geoff Romer, German Diago, Jan Moeller, Jonathan Wakely, Josh Berne, Kilian

Henneberger, LanguageLawyer, Lewis Baker, Louis Dionne, Maciej Bogus, Malcolm Parsons, Matthew Calabrese, Nathan Myers, Neelofer Banglawala, Nevin Liber, Nina Ranns, Patrice Roy, Roger Orr, Rostislav Khlebnikov, Stephan T. Lavavej, Stephen Kelly, Thomas Koeppe, Thomas Russell, Tom Hudson, Tomasz Kaminski, Tony van Eerd and Ville Voutilainen for suggestions and useful discussion.

References

A Preliminary Proposal for a Deep-Copying Smart Pointer
W. E. Brown, 2012
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3339.pdf

A polymorphic value-type for C++
J. B. Coe, S. Parent 2019
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0201r6.html

A Free-Store-Allocated Value Type for C++
J. B. Coe, A. Peacock 2022
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1950r2.html

An allocator-aware optional type
P. Halpern, N. D. Ranns, V. Voutilainen, 2024
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2047r7.html

MISRA Language Guidelines
https://ldra.com/misra/

High Integrity C++
https://www.perforce.com/resources/qac/high-integrity-cpp-coding-standard

Appendix A: Detailed design decisions

We discuss some of the decisions that were made in the design of `indirect` and `polymorphic`. Where there are multiple options, we discuss the advantages and disadvantages of each.

Two class templates, not one

It is conceivable that a single class template could be used as a vocabulary type for an indirect value type supporting polymorphism. However, implementing this would impose efficiency costs on the copy constructor when the owned object is the same type as the template type. When the owned object is a derived type, the copy constructor uses type erasure to perform dynamic dispatch and call the derived type copy constructor. The overhead of indirection and a virtual function call is not tolerable where the owned object type and template type match.

One potential solution would be to use a `std::variant` to store the owned type or the control block used to manage the owned type. This would allow the copy constructor to be implemented efficiently when the owned type and template type match. This would increase the object size beyond that of a single pointer as the discriminant must be stored.

For the sake of minimal size and efficiency, we opted to use two class templates.

Copiers, deleters, pointer constructors, and allocator support

The older types `indirect_value` and `polymorphic_value` had constructors that take a pointer, copier, and deleter. The copier and deleter could be used to specify how the object should be copied and deleted. The existence of a pointer constructor introduces undesirable properties into the design of `polymorphic_value`, such as allowing the possibility of object slicing on copy when the dynamic and static types of a derived-type pointer do not match.

We decided to remove the copier, delete, and pointer constructor in favour of adding allocator support. A pointer constructor and support for custom copiers and deleters are not core to the design of either class template; both could be added in a later revision of the standard if required.

We have been advised that allocator support must be a part of the initial implementation and cannot be added retrospectively. As `indirect` and `polymorphic` are intended to be used alongside other C++ standard library types, such as `std::map` and `std::vector`, it is important that they have allocator support in contexts where allocators are used.

Pointer-like helper functions

Earlier revisions of `polymorphic_value` had helper functions to get access to the underlying pointer. These were removed under the advice of the Library Evolution Working Group as they were not core to the design of the class template, nor were they consistent with value-type semantics.

Pointer-like accessors like `dynamic_pointer_cast` and `static_pointer_cast`, which are provided for `std::shared_ptr`, could be added in a later revision of the standard if required.

Constraints and incomplete type support

We decided not to conditionally enable the default constructor, copy constructor and comparison operators for `indirect`, and not to conditionally enable the default constructor for `polymorphic`.

Both `indirect` and `polymorphic` must support incomplete types at class instantiation time. Similar to `vector`, they may falsely advertise support (through type traits or concepts) for functions that would make a program ill-formed if they were used.

```
struct Copyable {
  Copyable() = default;
  Copyable(const Copyable&) = default;
};

struct NonCopyable {
  NonCopyable() = default;
  NonCopyable(const NonCopyable&) = delete;
};

struct Incomplete;

static_assert(std::is_copy_constructible_v<std::vector<Copyable>>); // Passes.
static_assert(std::is_copy_constructible_v<std::vector<NonCopyable>>); // Passes.
static_assert(std::is_copy_constructible_v<std::vector<Incomplete>>); // Passes.

static_assert(std::is_copy_constructible_v<indirect<Copyable>>); // Passes.
static_assert(std::is_copy_constructible_v<indirect<NonCopyable>>); // Passes.
static_assert(std::is_copy_constructible_v<indirect<Incomplete>>); // Passes.

static_assert(std::is_copy_constructible_v<polymorphic<Copyable>>); // Passes.
static_assert(std::is_copy_constructible_v<polymorphic<NonCopyable>>); // Passes.
static_assert(std::is_copy_constructible_v<polymorphic<Incomplete>>); // Passes.
```

Deferred constraints for incomplete types

It is possible to avoid misleading type trait information by using *deferred constraints*: constraint evaluation can be deferred to function instantiation time by applying constraints to a deduced template argument.

```
template <typename T>
class incomplete_wrapper {
  T* t; // Use a pointer for incomplete type support.

 public:
  template <typename TT=T>
  incomplete_wrapper()
  requires std::is_default_constructible_v<TT>;
  incomplete_wrapper(const incomplete_wrapper&) requires false;

  template <typename TT=T>
  incomplete_wrapper(const incomplete_wrapper&)
  requires std::is_copy_constructible_v<TT>;
};
```

Whilst appealing, this approach is problematic when used in composition with other types. To illustrate, consider the following class template `TaggedType`:

```
template<typename T>
class TaggedType {
  T t;

 public:
  TaggedType() requires std::is_default_constructible_v<T>;
  TaggedType(const TaggedType&) requires std::is_copy_constructible_v<T>;
};
```

We cannot instantiate the class `TaggedType<incomplete_wrapper<T>>` when `T` is an incomplete type as the constraints on `TaggedType` are evaluated at class instantiation time. This will force the evaluation of the constraints on `incomplete_wrapper<T>`, which cannot be evaluated as `T` is incomplete.

To support `incomplete_wrapper` with an incomplete type, `TaggedType` must also use deferred constraints. Any type with constraints that might be templated on `TaggedType` must then also use deferred constraints, and so on. The viral nature of this requirement makes `incomplete_wrapper` unusable with other library components if it is intended to support incomplete types.

We decided not to impose deferred constraints on `indirect` and `polymorphic`, as it would make the types unusable in composition with other types.

Mandates not constraints

`indirect` and `polymorphic` use static assertions (mandates clauses) which are evaluated at function instantiation time to provide better compiler errors.

For example, the code sample below:

```
indirect<NonCopyable> copy(const indirect<NonCopyable>& i) {
  return i;
}
```

gives the following errors:

```
indirect.h:LINE:COLUMN: error: static assertion failed
  LINE |     static_assert(std::copy_constructible<T>);

note: in instantiation of member function 'indirect<NonCopyable>::indirect' requested here
 LINE |   return i;
```

Arthur O'Dwyer has written a blog post on the topic of constraints and incomplete types https://quuxplusone.github.io/blog/2020/02/05/vector-is-copyable-except-when-its-not.

Implicit conversions

We decided that there should be no implicit conversion of a value `T` to an `indirect<T>` or `polymorphic<T>`. An implicit conversion would require using a memory resource and memory allocation, which is best made explicit by the user.

```
Rectangle r(w, h);
polymorphic<Shape> s = r; // error
```

To transform a value into `indirect` or `polymorphic`, the user must use the appropriate constructor.

```
Rectangle r(w, h);
polymorphic<Shape> s(std::in_place_type<Rectangle>, r);
assert(dynamic_cast<Rectangle*>(&*s) != nullptr);
```

Explicit conversions

The older class template `polymorphic_value` had explicit conversions, allowing construction of a `polymorphic_value<T>` from a `polymorphic_value<U>`, where `T` was a base class of `U`.

```
polymorphic_value<Quadrilateral> q(std::in_place_type<Rectangle>, w, h);
polymorphic_value<Shape> s = q;
assert(dynamic_cast<Rectangle*>(&*s) != nullptr);
```

Similar code cannot be written with `polymorphic` as it does not allow conversions between derived types:

```
polymorphic<Quadrilateral> q(std::in_place_type<Rectangle>, w, h);
polymorphic<Shape> s = q; // error
```

This is a deliberate design decision. `polymorphic` is intended to be used for ownership of member data in composite classes where compiler-generated special member functions will be used.

There is no motivating use case for explicit conversion between derived types outside of tests.

A converting constructor could be added in a future version of the C++ standard.

Comparisons for `indirect`

We implement comparisons for `indirect` in terms of `operator==` and `operator<=>` returning `bool` and `auto` respectively.

The alternative would be to implement the full suite of comparison operators, forwarding them to the underlying type and allowing non-boolean return types. Support for non-boolean return types would support unusual (non-regular) user-defined comparison operators which could be helpful when the underlying type is part of a domain-specific-language (DSL) that uses comparison operators for a different purpose. However, this would be inconsistent with other standard library types like `optional`, `variant` and `reference_wrapper`. Moreover, we'd likely give only partial support for a theoretical DSL which may well make use of other operators like `operator+` and `operator-` which are not supported for `indirect`.

Supporting `operator()` `operator[]`

There is no need for `indirect` or `polymorphic` to provide a function call or an indexing operator. Users who wish to do that can simply access the value and call its operator. Furthermore, unlike comparisons, function calls or indexing operators do not compose further; for example, a composite would not be able to automatically generate a composited `operator()` or an `operator[]`.

Supporting arithmetic operators

While we could provide support for arithmetic operators, `+`, `-` ,`*`, `/`, to `indirect` in the same way that we support comparisons, we have chosen not to do so. The arithmetic operators would need to support a valueless state for which there is no precedent in the standard library.

Support for arithmetic operators could be added in a future version of the C++ standard. If support for arithmetic operators for valueless or empty objects is later added to the standard library in a coherent way, it could be added for `indirect` at that time.

Member function `emplace`

Neither `indirect` nor `polymorphic` support `emplace` as a member function. The member function `emplace` could be added as :

```
template <typename ...Ts>
indirect::emplace(Ts&& ...ts);
```

```
template <typename U, typename ...Ts>
polymorphic::emplace(in_place_type<U>, Ts&& ...ts);
```

This would be API noise. It offers no efficiency improvement over:

```
some_indirect = indirect(/* arguments */);
```

```
some_polymorphic = polymorphic(in_place_type<U>, /* arguments */);
```

Support for an emplace member function could be added in a future version of the C++ standard.

Small Buffer Optimisation

It is possible to implement `polymorphic` with a small buffer optimisation, similar to that used in `std::function`. This would allow `polymorphic` to store small objects without allocating memory. Like `std::function`, the size of the small buffer is left to be specified by the implementation.

The authors are sceptical of the value of a small buffer optimisation for objects from a type hierarchy. If the buffer is too small, all instances of `polymorphic` will be larger than needed. This is because they will allocate memory in addition to having the memory from the (empty) buffer as part of the object size. If the buffer is too big, `polymorphic` objects will be larger than necessary, potentially introducing the need for `indirect<polymorphic<T>>`.

We could add a non-type template argument to `polymorphic` to specify the size of the small buffer:

```
template <typename T, typename Alloc, size_t BufferSize>
class polymorphic;
```

However, we opt not to do this to maintain consistency with other standard library types. Both `std::function` and `std::string` leave the buffer size as an implementation detail. Including an additional template argument in a later revision of the standard would be a breaking change. With usage experience, implementers will be able to determine if a small buffer optimisation is worthwhile, and what the optimal buffer size might be.

A small buffer optimisation makes little sense for `indirect` as the sensible size of the buffer would be dictated by the size of the stored object. This removes support for incomplete types and locates storage for the object locally, defeating the purpose of `indirect`.

Appendix B: Before and after examples

We include some minimal, illustrative examples of how `indirect` and `polymorphic` can be used to simplify composite class design.

Using `indirect` for binary compatibility using the PIMPL idiom

Without `indirect`, we use `std::unique_ptr` to manage the lifetime of the implementation object. All const-qualified methods of the composite will need to be manually checked to ensure that they are not calling non-const qualified methods of component objects.

Before, without using `indirect`

```
// Class.h

class Class {
  class Impl;
  std::unique_ptr<Impl> impl_;
 public:
  Class();
  ~Class();
  Class(const Class&);
  Class& operator=(const Class&);
```

```cpp
  Class(Class&&) noexcept;
  Class& operator=(Class&&) noexcept;

  void do_something();
};
// Class.cpp

class Impl {
 public:
  void do_something();
};

Class::Class() : impl_(std::make_unique<Impl>()) {}

Class::~Class() = default;

Class::Class(const Class& other) : impl_(std::make_unique<Impl>(*other.impl_)) {}

Class& Class::operator=(const Class& other) {
  if (this != &other) {
    Class tmp(other);
    using std::swap;
    swap(*this, tmp);
  }
  return *this;
}

Class(Class&&) noexcept = default;
Class& operator=(Class&&) noexcept = default;

void Class::do_something() {
  impl_->do_something();
}
```

After, using `indirect`

```cpp
// Class.h

class Class {
  indirect<class Impl> impl_;
 public:
  Class();
  ~Class();
  Class(const Class&);
  Class& operator=(const Class&);
  Class(Class&&) noexcept;
  Class& operator=(Class&&) noexcept;

  void do_something();
};
// Class.cpp

class Impl {
 public:
  void do_something();
};
```

```
Class::Class() : impl_(indirect<Impl>()) {}
Class::~Class() = default;
Class::Class(const Class&) = default;
Class& Class::operator=(const Class&) = default;
Class(Class&&) noexcept = default;
Class& operator=(Class&&) noexcept = default;

void Class::do_something() {
  impl_->do_something();
}
```

Using `polymorphic` for a composite class

Without `polymorphic`, we use `std::unique_ptr` to manage the lifetime of component objects. All const-qualified methods of the composite will need to be manually checked to ensure that they are not calling non-const qualified methods of component objects.

Before, without using `polymorphic`

```
class Canvas;

class Shape {
 public:
  virtual ~Shape() = default;
  virtual std::unique_ptr<Shape> clone() = 0;
  virtual void draw(Canvas&) const = 0;
};

class Picture {
  std::vector<std::unique_ptr<Shape>> shapes_;

 public:
  Picture(const std::vector<std::unique_ptr<Shape>>& shapes) {
    shapes_.reserve(shapes.size());
    for (auto& shape : shapes) {
      shapes_.push_back(shape->clone());
    }
  }

  Picture(const Picture& other) {
    shapes_.reserve(other.shapes_.size());
    for (auto& shape : other.shapes_) {
      shapes_.push_back(shape->clone());
    }
  }

  Picture& operator=(const Picture& other) {
    if (this != &other) {
      Picture tmp(other);
      using std::swap;
      swap(*this, tmp);
    }
    return *this;
  }

  void draw(Canvas& canvas) const;
};
```

After, using `polymorphic`

```
class Canvas;

class Shape {
 protected:
  ~Shape() = default;

 public:
  virtual void draw(Canvas&) const = 0;
};

class Picture {
  std::vector<polymorphic<Shape>> shapes_;

 public:
  Picture(const std::vector<polymorphic<Shape>>& shapes)
      : shapes_(shapes) {}

  // Picture(const Picture& other) = default;

  // Picture& operator=(const Picture& other) = default;

  void draw(Canvas& canvas) const;
};
```

Appendix C: Design choices, alternatives and breaking changes

The table below shows the main design components considered, the key design decisions made, and the cost and impact of alternative design choices. As presented in this paper, the design of class templates `indirect` and `polymorphic` has been approved by the LEWG. The authors have until C++26 is standardized to consider making any breaking changes; after C++26, whilst breaking changes will still be possible, the impact of these changes on users could be potentially significant and unwelcome.

Component	Decision	Alternative	Change impact	Breaking change?
Member `emplace`	No member `emplace`	Add member `emplace`	Pure addition	No
`operator bool`	No `operator bool`	Add `operator bool`	Changes semantics	No
`indirect` comparsion preconditions	Allow comparison of valueless objects	`indirect` must not be valueless	Previously valid code would invoke undefined behaviour	Yes
`indirect` hash preconditions	Allow hash of valueless objects	`indirect` must not be valueless	Previously valid code would invoke undefined behaviour	Yes
Copy and copy assign preconditions	Object can be valueless	Forbids copying of valueless objects	Previously valid code would invoke undefined behaviour	Yes

Component	Decision	Alternative	Change impact	Breaking change?
Move and move assign preconditions	Object can be valueless	Forbids moving of valueless objects	Previously valid code would invoke undefined behaviour	Yes
Requirements on `T` in `polymorphic<T>`	No requirement that `T` has virtual functions	Add *Mandates* to require `T` to have virtual functions	Code becomes ill-formed	Yes
State of default-constructed object	Default-constructed object (where valid) has a value	Make default-constructed object valueless	Changes semantics; necessitates adding `operator bool` and allowing move, copy and compare of valueless (empty) objects	Yes
Small buffer optimisation for polymorphic	SBO is not required, settings are hidden	Add buffer size and alignment as template parameters	Breaks ABI; forces implementers to use SBO	Yes
`noexcept` for accessors	Accessors are `noexcept` like `unique_ptr` and `optional`	Remove `noexcept` from accessors	User functions marked `noexcept` could be broken	Yes
Specialization of optional	No specialization of optional	Specialize optional to use valueless state	Breaks ABI; engaged but valueless optional would become indistinguishable from a disengaged optional	Yes
Permit user specialization	No user specialization is permitted	Permit specialization for user-defined types	Previously ill-formed code would become well-formed	No
Explicit constructors	Constructors are marked `explicit`	Non-explicit constructors	Conversion for single arguments or braced initializers becomes valid	No
Support comparisons for indirect	Comparisons are supported when the owned type supports them	No support for comparisons	Previously valid code would become ill-formed	Yes
Support arithmetic operations for `indirect`	No support for arithmetic operations	Forward arithemtic operations to the owned type when it supports them	Previously ill-formed code would become well-formed	No

Component	Decision	Alternative	Change impact	Breaking change?
Support `operator ()` for `indirect`	No support for `operator ()`	Forward `operator()` to the owned type when it is supported	Previously ill-formed code would become well-formed	No
Support `operator []` for `indirect`	No support for `operator []`	Forward `operator[]` to the owned type when it is supported	Previously ill-formed code would become well-formed	No