`copy_on_write`: A Vocabulary Type for Lazily-Copied Values

ISO/IEC JTC1 SC22 WG21 Programming Language C++

P4210R0

Working Group: Library Evolution, Library

Date: 2026-05-05

Daniel Pfeifer <daniel@pfeifer-mail.de>

Jonathan Coe <jonathanbcoe@gmail.com>

Antony Peacock <ant.peacock@gmail.com>

Abstract

We propose the addition of a new class template `copy_on_write<T>` to the C++ Standard Library.

`copy_on_write<T>` manages a dynamically-allocated owned object of type `T` with shared, reference-counted ownership. Copies of a `copy_on_write<T>` share ownership of the same underlying object; the object is copied only when a modification is requested and the reference count exceeds one. This gives efficient value semantics: copying is O(1) regardless of the size or complexity of `T`, and mutation is deferred until necessary.

Specializations of `copy_on_write` have value semantics and compose well with other standard library types such as `vector`, allowing the compiler to correctly generate special member functions.

Access to the owned object through `operator*` and `operator->` is always const-qualified. Mutation of the owned object is performed exclusively through the `modify` member function, which copies the shared data when necessary and provides a structured interface for efficient in-place or constructive modification.

Motivation

Expensive Copies in Composite Classes

When composite objects contain large or expensive-to-copy members, value semantics become costly. Consider a text editor that maintains a document as a sequence of lines:

```
class document {
  std::vector<std::string> _lines;
public:
```

```
  document() = default;
  void append(std::string line) {
    _lines.push_back(std::move(line));
  }
  std::size_t line_count() const { return _lines.size(); }
};
```

If `document` objects are copied frequently (passed by value, returned from functions, stored in containers) but modified rarely, eagerly copying all lines on every copy is unnecessarily expensive. Sharing the underlying storage (e.g. with `std::shared_ptr`) avoids copies but can compromise value semantics.

`copy_on_write<T>` provides a middle ground. Multiple objects share the underlying data, paying only an atomic reference-count increment on copy. A copy of the data is made only when one of the sharing objects needs to mutate it.

Composing Cheaply-Copyable Types

`copy_on_write<T>` allows composite classes to be cheaply copyable even when their components are expensive to copy:

```
class document {
  std::copy_on_write<std::vector<std::string>> _lines;
public:
  document() = default;

  // Copying is O(1): just an atomic increment.
  document(const document&) noexcept = default;
  document& operator=(const document&) noexcept = default;

  // Mutation copies the underlying data only if shared.
  void append(std::string line) {
    _lines.modify([&](std::vector<std::string>& v) {
      v.push_back(std::move(line));
    });
  }

  std::size_t line_count() const { return _lines->size(); }
};
```

The class above has full value semantics — compiler-generated copy, move, and assignment work correctly — while avoiding expensive deep copies when the document is only being read.

Efficient Fork-and-Edit Patterns

`copy_on_write<T>` is well suited to fork-and-edit workflows where a derived value is created from an existing one with few changes:

```
std::copy_on_write<Config> cfg = base_config();
std::copy_on_write<Config> debug_cfg(cfg);  // O(1), shared
debug_cfg.modify([](Config& c) {
  c.set_log_level(LogLevel::Debug);
});
// cfg is unchanged; debug_cfg holds a private copy.
```

Without `copy_on_write`, the second line would require a full deep copy of `Config` regardless of whether it is ever modified.

Avoiding Redundant Work on Shared Data

When two `copy_on_write<T>` objects share the underlying data, they are provably equal. The equality comparison short-circuits in this case, avoiding potentially expensive element-wise comparisons.

Design Requirements

Shared Ownership with Value Semantics

Unlike `indirect<T>`, which has unique ownership, `copy_on_write<T>` uses shared, reference-counted ownership. The reference count is maintained atomically to allow copies to be safely shared across threads.

Despite shared ownership of the underlying storage, `copy_on_write<T>` presents value semantics to its users: two `copy_on_write<T>` objects are equal when their owned objects are equal, and modifying one does not affect the other.

A new type rather than a composable wrapper

If copy-on-write semantics could be imparted to other types using an adapter class template, we could use `copy_on_write_adapter<indirect<T>>` or `copy_on_write_adapter<polymorphic<T>>` to create copy-on-write types. This adapter class could be used similarly to container adapters such as `queue`, `stack` and `priority_queue`.

Implementing copy-on-write behaviour requires intrusion into the details of object ownership and is not possible to implement for `indirect<T>` or `polymorphic<T>` without adding an additional layer of indirection.

To support polymorphic copy-on-write behaviour, an additional class, `polymorphic_copy_on_write` would be required. We are not proposing addition of this class template in this proposal.

Cheap Copies

Copying a `copy_on_write<T>` increments the reference count and shares a pointer to the owned object. The cost is O(1) and independent of the size

or complexity of `T`. The owned object is copied only when `modify` is called and the reference count exceeds one.

Read-Only Access via Observers

`copy_on_write<T>` provides only const access to the owned object through `operator*` and `operator->`. Mutable access is not available directly. This invariant is necessary for correctness: if a mutable reference to the shared object were handed out without a preceding copy, modifications through that reference would be visible to all other objects sharing the same data, violating value semantics.

When a `copy_on_write<T>` is accessed through a const access path, constness propagates to the owned object. Because all access is already const-qualified, this is automatically satisfied.

Modification Through `modify`

Modification of the owned object is performed through the `modify` member function. Two overloads are provided:

`modify(action)` accepts a callable of the form `void(T&)`. If the reference count is one (exclusive ownership), the action is applied directly to the owned object. If the reference count exceeds one (shared ownership), the owned object is first copied, ownership becomes exclusive, and then the action is applied.

`modify(action, transform)` accepts both a `void(T&)` action and a `T(T const&)` transformation. If the reference count is one, the action is applied in-place. If the reference count exceeds one, a new owned object is constructed by calling the transformation on the current value, without first copying the old value. This allows the caller to provide an optimised construction path for the shared case.

For example, when inserting an element at the front of a vector: the in-place action calls `insert` on the existing vector; the transformation constructs a new vector from scratch, placing the new element first and then appending the remaining elements. In the shared case this avoids the cost of first copying the full vector and then inserting.

Identity Optimisation

Two `copy_on_write<T>` objects that share the same underlying object are provably equal. The `identical_to` member function returns `true` when two objects of the same specialisation share the same owned object. Equality comparison between two objects of the same specialisation short-circuits to `true` when `identical_to` returns `true`, avoiding a potentially expensive value comparison.

Users may also call `identical_to` directly to short-circuit work in algorithms that operate on `copy_on_write<T>` values.

The Valueless State and Interaction with `std::optional`

`copy_on_write<T>` has a valueless state used to implement move. A `copy_on_write<T>` object becomes valueless only after it has been moved from. Accessing the owned object of a valueless object is undefined behaviour. The `valueless_after_move` member function returns `true` if the object is in the valueless state.

The valueless state is not intended to be observable in normal use. There is no `operator bool` or `has_value` member function.

Where a nullable `copy_on_write<T>` is required, `std::optional<copy_on_write<T>>` is recommended.

Allocator Support

`copy_on_write<T>` is allocator-aware. The allocator is used to allocate and deallocate storage for an internal control block that holds both the reference count and the owned object.

When two `copy_on_write<T>` objects have equal allocators, a copy shares the underlying data; when allocators differ, a deep copy is made. This ensures that an object created with one memory resource is not owned by another.

Special Member Functions

`copy_on_write<T>` is suitable for use as a member of composite classes where the compiler will generate special member functions. Copy construction and copy assignment are available whenever `T` is copy constructible. Move construction is unconditionally available and noexcept.

Prior Work

The copy-on-write idiom has a long history in C++. Qt's `QSharedDataPointer` (Qt 4.0, 2005) pioneered a reusable COW pointer type in a major C++ framework. Adobe's `stlab` library provides a `copy_on_write` type along similar lines. The GCC implementation of `std::basic_string` prior to C++11 used copy-on-write semantics internally; it was later removed due to incompatibilities with the C++11 memory model and iterator invalidation requirements.

This proposal standardises a modern, allocator-aware formulation aligned with the design of `std::indirect` [P3019].

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 macro with an editor-provided value to [version.syn]:

```
#define __cpp_lib_copy_on_write ??????L  // also in <memory>
```

Header <memory> synopsis [memory]

Add the following to the <memory> synopsis within namespace std:

```
// [cow], class template copy_on_write
template <class T, class Allocator = allocator<T>>
  class copy_on_write;

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

namespace pmr {
  template <class T>
    using copy_on_write =
      std::copy_on_write<T, polymorphic_allocator<T>>;
}
```

X.Y Class template copy_on_write [cow]

X.Y.1 General [cow.general]

- A copy_on_write object manages the lifetime of an owned object. A copy_on_write object is *valueless* if it has no owned object. A copy_on_write object may become valueless only after it has been moved from.
- Multiple copy_on_write objects may share ownership of the same owned object. A reference count is maintained to track the number of objects sharing ownership. The reference count is modified atomically. When the reference count reaches zero, the owned object is destroyed and its storage is deallocated.
- In every specialization copy_on_write<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 copy_on_write<T, Allocator> uses an object of type Allocator to allocate and free storage for the owned object as needed.

- A program that instantiates the definition of the template `copy_on_write<T, Allocator>` with a non-object type, an array type, `in_place_t`, a specialization of `in_place_type_t`, or a cv-qualified type for the `T` parameter is ill-formed.
- The template parameter `T` of `copy_on_write` may be an incomplete type.
- The template parameter `Allocator` of `copy_on_write` shall meet the *Cpp17Allocator* requirements.
- If a program declares an explicit or partial specialization of `copy_on_write`, the behavior is undefined.

The following exposition-only concepts are used in the specification. Let `F` be a type and `T` a value type.

- *`cow-action<F, T>`* is satisfied when `F` is invocable with an lvalue of type `T` and the result type is `void`.
- *`cow-transformation<F, T>`* is satisfied when `F` is invocable with a const lvalue of type `T` and the result type is `T`.

X.Y.2 Synopsis [cow.syn]

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

  explicit copy_on_write();

  explicit copy_on_write(
    allocator_arg_t, const Allocator& a);

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

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

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

  template <class... Us>
  explicit copy_on_write(
```

```
  allocator_arg_t, const Allocator& a,
  in_place_t, Us&&... us);

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

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

explicit copy_on_write(
  allocator_arg_t, const Allocator& a,
  const copy_on_write& other);

explicit copy_on_write(
  allocator_arg_t, const Allocator& a,
  copy_on_write&& other)
  noexcept(
    allocator_traits<Allocator>::is_always_equal::value);

copy_on_write(const copy_on_write& other);

copy_on_write(copy_on_write&& other) noexcept;

~copy_on_write();

copy_on_write& operator=(const copy_on_write& other);

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

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

const T& operator*() const noexcept;

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

```
  bool valueless_after_move() const noexcept;

  allocator_type get_allocator() const noexcept;

  long use_count() const noexcept;

  bool identical_to(
    const copy_on_write& other) const noexcept;

  template <class F>  // requires cow-action<F, T>
  void modify(F&& f);

  template <class F, class G>
    // requires cow-action<F, T>
    //       && cow-transformation<G, T>
  void modify(F&& f, G&& g);

  void swap(copy_on_write& other) noexcept(see below);

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


template <class T1, class A1, class T2, class A2>
bool operator==(
  const copy_on_write<T1, A1>& lhs,
  const copy_on_write<T2, A2>& rhs)
  noexcept(see below);

template <class T, class A, class U>
bool operator==(
  const copy_on_write<T, A>& lhs,
  const U& rhs) noexcept(see below);

template <class T1, class A1, class T2, class A2>
auto operator<=>(
  const copy_on_write<T1, A1>& lhs,
  const copy_on_write<T2, A2>& rhs)
  -> synth-three-way-result<T1, T2>;

template <class T, class A, class U>
auto operator<=>(
  const copy_on_write<T, A>& lhs,
  const U& rhs)
```

```
  -> synth-three-way-result<T, U>;

template <class T, class A>
void swap(
  copy_on_write<T, A>& lhs,
  copy_on_write<T, A>& rhs)
  noexcept(noexcept(lhs.swap(rhs)));

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

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

X.Y.3 Constructors [cow.ctor]

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

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

```
explicit copy_on_write();
```

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

```
explicit copy_on_write(allocator_arg_t, const Allocator& a);
```

- *Mandates*: `is_default_constructible_v<T>` is `true`.
- *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`.

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

- *Constraints*:
  - `is_same_v<remove_cvref_t<U>, copy_on_write>` 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`.
- *Effects*: Constructs an owned object of type `T` with `std::forward<U>(u)`, using the allocator `alloc`.

```
template <class U = T>
explicit copy_on_write(
```

```
  allocator_arg_t, const Allocator& a, U&& u);
```

- *Constraints*:
  - `is_same_v<remove_cvref_t<U>, copy_on_write>` is `false`,
  - `is_same_v<remove_cvref_t<U>, in_place_t>` is `false`, and
  - `is_constructible_v<T, U>` is `true`.
- *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 copy_on_write(in_place_t, Us&&... us);
```

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

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

- *Constraints*: `is_constructible_v<T, Us...>` is `true`.
- *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 copy_on_write(
  in_place_t,
  initializer_list<I> ilist, Us&&... us);
```

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

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

- *Constraints*: `is_constructible_v<T, initializer_list<I>&, Us...>` is `true`.
- *Effects*: `alloc` is direct-non-list-initialized with `a`. Constructs an owned

object of type `T` with `ilist, std::forward<Us>(us)...`, using the allocator `alloc`.

```
explicit copy_on_write(
  allocator_arg_t, const Allocator& a,
  const copy_on_write& other);
```

- *Mandates*: `is_copy_constructible_v<T>` is `true`.
- *Effects*: `alloc` is direct-non-list-initialized with `a`. If `other` is valueless, `*this` is valueless. Otherwise, if `a == other.alloc` is `true`, `*this` shares ownership of `other`'s owned object and the reference count is incremented. Otherwise, constructs an owned object of type `T` with `*other`, using the allocator `alloc`.

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

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

```
copy_on_write(const copy_on_write& other);
```

- *Mandates*: `is_copy_constructible_v<T>` is `true`.
- *Effects*: `alloc` is direct-non-list-initialized with `allocator_traits<Allocator>::select_on_container` If `other` is valueless, `*this` is valueless. Otherwise, if `alloc == other.alloc` is `true`, `*this` shares ownership of `other`'s owned object and the reference count is incremented. Otherwise, constructs an owned object of type `T` with `*other`, using the allocator `alloc`.

```
copy_on_write(copy_on_write&& other) noexcept;
```

- *Effects*: `alloc` is direct-non-list-initialized with `other.alloc`. If `other` is valueless, `*this` is valueless. Otherwise, `*this` takes exclusive ownership of the owned object of `other`.
- *Postconditions*: `other` is valueless.

X.Y.4 Destructor [cow.dtor]

```
~copy_on_write();
```

- *Mandates*: `T` is a complete type.
- *Effects*: Decrements the reference count of the owned object. If the reference count reaches zero, destroys the owned object using

`allocator_traits<Allocator>::destroy` and deallocates the storage.

X.Y.5 Assignment [cow.assign]

```
copy_on_write& operator=(const copy_on_write& other);
```

- *Mandates*: `is_copy_constructible_v<T>` is `true`.
- *Effects*: If `addressof(other) == this` is `true`, there are no effects. Otherwise:
  1. The allocator needs updating if `allocator_traits<Allocator>::propagate_on_container_copy_` is `true`.
  2. If `other` is valueless, `*this` becomes valueless. The reference count of the previously-owned object in `*this`, if any, is decremented; if it reaches zero, the owned object is destroyed and storage is deallocated.
  3. Otherwise, if the allocator does not need updating and `alloc == other.alloc` is `true`, `*this` shares ownership of `other`'s owned object. The reference count is incremented.
  4. Otherwise, constructs a new owned object of type `T` with `*other`, using the new allocator.
  5. The reference count of the previously-owned object in `*this`, if any, is decremented; if it reaches zero, the owned object is destroyed and storage is deallocated.
  6. If the allocator needs updating, the allocator in `*this` is replaced with a copy of the allocator in `other`.
- *Returns*: A reference to `*this`.

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

- *Effects*: If `addressof(other) == this` is `true`, there are no effects. Otherwise:
  1. The allocator needs updating if `allocator_traits<Allocator>::propagate_on_container_move_` is `true`.
  2. If `other` is valueless, `*this` becomes valueless. The reference count of the previously-owned object in `*this`, if any, is decremented; if it reaches zero, the owned object is destroyed and storage is deallocated.
  3. Otherwise, if the allocator does not need updating and `alloc == other.alloc` is `true`, `*this` takes exclusive ownership of the owned object of `other`.

4. Otherwise, constructs a new owned object with `std::move(*other)`, using the allocator `alloc`.
5. The reference count of the previously-owned object in `*this`, if any, is decremented; if it reaches zero, the owned object is destroyed and storage is deallocated.
6. If the allocator needs updating, the allocator in `*this` is replaced with a copy of the allocator in `other`.

- *Postconditions*: `other` is valueless.
- *Returns*: A reference to `*this`.

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

- *Constraints*:
  - `is_same_v<remove_cvref_t<U>, copy_on_write>` is `false`,
  - `is_constructible_v<T, U>` is `true`, and
  - `is_assignable_v<T&, U>` is `true`.
- *Effects*: If `*this` is not valueless and `use_count() == 1`, equivalent to `**this = std::forward<U>(u)`. Otherwise, constructs a new `copy_on_write<T>` from `std::forward<U>(u)` and move-assigns it to `*this`.
- *Returns*: A reference to `*this`.

X.Y.6 Observers [cow.obs]

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

- *Preconditions*: `*this` is not valueless.
- *Returns*: A const reference to the owned object.

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

- *Preconditions*: `*this` is not valueless.
- *Returns*: A pointer to the owned object.

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

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

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

- *Returns*: `alloc`.

```
long use_count() const noexcept;
```

- *Preconditions*: `*this` is not valueless.
- *Returns*: The number of `copy_on_write` objects currently sharing ownership of the owned object.

- *Remarks*: In the presence of concurrent access, the returned value may be stale by the time it is observed. It should be treated as an optimisation hint rather than a precise count.

```
bool identical_to(const copy_on_write& other) const noexcept;
```

- *Preconditions*: Neither `*this` nor `other` is valueless.
- *Returns*: `true` if `*this` and `other` share the same underlying owned object, otherwise `false`.

X.Y.7 Modifiers [cow.mod]

```
template <class F>  // requires cow-action<F, T>
void modify(F&& f);
```

- *Constraints*: *cow-action<F, T>* is satisfied.
- *Preconditions*: `*this` is not valueless.
- *Effects*: If `use_count() == 1`, calls `std::forward<F>(f)` with a mutable reference to the owned object. Otherwise, creates a new owned object as a copy of the current owned object (using the allocator `alloc`), decrements the reference count of the previous owned object, takes exclusive ownership of the new object, and calls `std::forward<F>(f)` with a mutable reference to the new owned object.

```
template <class F, class G>
  // requires cow-action<F, T> && cow-transformation<G, T>
void modify(F&& f, G&& g);
```

- *Constraints*: *cow-action<F, T>* and *cow-transformation<G, T>* are both satisfied.
- *Preconditions*: `*this` is not valueless.
- *Effects*: If `use_count() == 1`, calls `std::forward<F>(f)` with a mutable reference to the owned object. Otherwise, creates a new owned object by calling `std::forward<G>(g)` with a const reference to the current owned object (using the allocator `alloc`), decrements the reference count of the previous owned object, and takes exclusive ownership of the new object.
- *Remarks*: In the shared case, the transformation `g` is responsible for constructing the new value. Unlike the single-argument `modify(action)` overload, the old owned object is not copied before the new one is created; `g` receives only a const reference. This allows callers to construct the new value more efficiently than copy-then-modify.

X.Y.8 Swap [cow.swap]

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

- *Preconditions*: If `allocator_traits<Allocator>::propagate_on_container_swap::value` is `false`, then `get_allocator() == other.get_allocator()` is `true`.
- *Effects*: Exchanges the owned objects or valueless states of `*this` and `other`. If `allocator_traits<Allocator>::propagate_on_container_swap::value` is `true`, the allocators of `*this` and `other` are exchanged. Otherwise the allocators are not swapped. [*Note*: Does not call `swap` on the owned objects directly. —*end note*]

```
template <class T, class A>
void swap(
  copy_on_write<T, A>& lhs,
  copy_on_write<T, A>& rhs)
  noexcept(noexcept(lhs.swap(rhs)));
```

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

X.Y.9 Relational operators [cow.relops]

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

- *Mandates*: The expression `*lhs == *rhs` is well-formed and its result is convertible to `bool`.
- *Returns*: If `lhs` is valueless or `rhs` is valueless, `lhs.valueless_after_move() == rhs.valueless_after_move()`. Otherwise, if `T`, `U`, `Allocator`, and `AA` are the same types and `lhs.identical_to(rhs)` is `true`, `true`. Otherwise, `*lhs == *rhs`.

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

- *Returns*: If `lhs` is valueless or `rhs` is valueless, `!lhs.valueless_after_move() <=> !rhs.valueless_after_move()`. Otherwise, if `T`, `U`, `Allocator`, and `AA` are the same types and `lhs.identical_to(rhs)` is `true`, `strong_ordering::equal`. Otherwise, `synth-three-way(*lhs, *rhs)`.

X.Y.10 Comparison with T [cow.comp.with.t]

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

- *Mandates*: The expression `*lhs == rhs` is well-formed and its result is convertible to `bool`. `U` is not a specialization of `copy_on_write`.
- *Returns*: If `lhs` is valueless, `false`; otherwise `*lhs == rhs`.

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

- *Mandates*: `U` is not a specialization of `copy_on_write`.
- *Returns*: If `lhs` is valueless, `strong_ordering::less`; otherwise `synth-three-way(*lhs, rhs)`.

X.Y.11 Hash support [cow.hash]

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

The specialization `hash<copy_on_write<T, Allocator>>` is enabled ([unord.hash]) if and only if `hash<T>` is enabled. When enabled, for an object `c` of type `copy_on_write<T, Allocator>`, `hash<copy_on_write<T, Allocator>>()(c)` evaluates to `hash<T>()(*c)` if `c` is not valueless, otherwise to an implementation-defined value. The member functions are not guaranteed to be noexcept.

Design Discussions

Read-Only Access and the `modify` Interface

The absence of a non-const `operator*` and `operator->` is the most distinctive aspect of `copy_on_write` compared to `indirect`.

Providing a `T&` directly would require one of two things: either eagerly copy the shared object at the point of access (eliminating the copy-on-write optimisation entirely), or return a `T&` to the shared object without copying. The latter would allow the caller to modify data visible to all other owners, violating value semantics.

The `modify` interface resolves this tension. By accepting a callable, it can determine at the point of invocation whether a copy is needed (when `use_count() > 1`) or whether in-place modification is safe (when `use_count() == 1`).

The two-argument overload `modify(action, transform)` further allows callers to provide a more efficient algorithm for the shared case. When ownership is not exclusive and the old object must be discarded, constructing the new value from scratch (via `transform`) is often more efficient than copy-constructing and then modifying. The caller is in the best position to provide this optimisation.

Thread Safety

The reference count is maintained with atomic operations, making it safe to copy and destroy `copy_on_write<T>` objects concurrently from different threads. However, concurrent reads of the owned object are safe only in the usual sense: multiple threads may call `operator*` and `operator->` simultaneously on objects sharing the same underlying data. Concurrent calls to `modify` on distinct objects that share data are safe; the first caller to modify will detach its copy, leaving the other objects undisturbed.

Concurrent calls to `modify` on the *same* `copy_on_write<T>` object, or concurrent read and write of the same object, require external synchronisation, as with any mutable standard library type.

Interaction with `shared_ptr`

`copy_on_write<T>` has a similar ownership model to `shared_ptr<const T>`: multiple owners, atomic reference counting. It differs in several important respects:

- The owned object is never accessible through a mutable pointer, eliminating the possibility of inadvertent mutation of shared state.
- Mutation is performed through the first-class `modify` interface rather than being entirely absent.
- `copy_on_write` has value semantics: it is copy-assignable and participates in value-based equality comparison and hashing.
- `copy_on_write` has no null state and does not support pointer arithmetic or custom deleters.
- `copy_on_write` is allocator-aware in the standard manner.

`use_count` and `identical_to`

`use_count` and `identical_to` expose internal sharing state and are provided primarily as optimisation hints, analogous to `shared_ptr::use_count`. Their values may change concurrently when other objects sharing the same data are copied or destroyed. Programs should not rely on the exact value of `use_count` for correctness in the presence of concurrent access.

`identical_to` is additionally used internally to provide an O(1) fast path in equality comparison: two objects sharing the same underlying data are provably equal without inspecting their values.

Explicit Constructors

All constructors of `copy_on_write` are marked `explicit`. Using dynamically-allocated shared storage is a deliberate choice that should not occur implicitly. This is consistent with the design of `indirect` and `polymorphic` [P3019].

Interaction with `indirect`

`copy_on_write<T>` and `indirect<T>` are complementary vocabulary types for dynamically-allocated values:

Property | `indirect<T>` | `copy_on_write<T>` |

|———————-|———-|———–| | Ownership | Unique | Shared (COW) | | Copy cost | O(sizeof T) | O(1) | | Mutable access | Yes | Via `modify` only | | `use_count` | No | Yes | | Equality short-circuit | No | Yes (identical_to) | | Const propagation | Yes | Yes (always const) | | Allocator-aware | Yes | Yes |

Prefer `indirect<T>` when mutation is frequent and sharing is not expected. Prefer `copy_on_write<T>` when copies are frequent, mutations are rare, or sharing can be exploited for efficiency.

Reference Implementation

A C++20 reference implementation is available at https://github.com/purpleKarrot/copy-on-write.

Acknowledgements

The authors would like to thank the authors of `stlab::copy_on_write` and Qt's `QSharedDataPointer` for pioneering this pattern. Thanks also to the authors of P3019 (`indirect` and `polymorphic`) for establishing the vocabulary type design conventions followed here.

References

Qt QSharedDataPointer https://doc.qt.io/qt-6/qshareddatapointer.html

`indirect` and `polymorphic`: *Vocabulary Types for Composite Class Design* J. B. Coe, A. Peacock, S. Parent, 2025 https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3019r14.html

stlab copy_on_write https://stlab.cc/libraries/copy_on_write.hpp/

Appendix A: Before and After Examples

We show how `copy_on_write` simplifies composite class design and reduces copy overhead in a text-editor document model.

Without `copy_on_write`

```
// document.h

class document {
  std::vector<std::string> _lines;
public:
  document() = default;

  // Deep copy required - O(number of lines).
  document(const document&) = default;
  document& operator=(const document&) = default;

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

  void append(std::string line) {
    _lines.push_back(std::move(line));
  }

  void insert(std::size_t pos, std::string line) {
    _lines.insert(_lines.begin() + pos, std::move(line));
  }

  void erase(std::size_t pos) {
    _lines.erase(_lines.begin() + pos);
  }

  std::size_t line_count() const { return _lines.size(); }

  const std::string& line(std::size_t i) const {
    return _lines[i];
  }
};
```

After, using `copy_on_write`

```
// document.h

class document {
  std::copy_on_write<std::vector<std::string>> _lines;
public:
  document() = default;

  // Copy is O(1): only an atomic reference-count increment.
  document(const document&) noexcept = default;
```

```
document& operator=(const document&) noexcept = default;

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

void append(std::string line) {
  _lines.modify(
    [&](std::vector<std::string>& v) {
      v.push_back(std::move(line));
    });
}

void insert(std::size_t pos, std::string line) {
  _lines.modify(
    // In-place: exclusive ownership.
    [&](std::vector<std::string>& v) {
      v.insert(v.begin() + pos, line);
    },
    // Constructive: shared ownership, no prior copy.
    [&](const std::vector<std::string>& v) {
      std::vector<std::string> w;
      w.reserve(v.size() + 1);
      w.insert(w.end(), v.begin(), v.begin() + pos);
      w.push_back(std::move(line));
      w.insert(w.end(), v.begin() + pos, v.end());
      return w;
    });
}

void erase(std::size_t pos) {
  _lines.modify(
    [&](std::vector<std::string>& v) {
      v.erase(v.begin() + pos);
    },
    [&](const std::vector<std::string>& v) {
      std::vector<std::string> w;
      w.reserve(v.size() - 1);
      w.insert(w.end(), v.begin(), v.begin() + pos);
      w.insert(w.end(), v.begin() + pos + 1, v.end());
      return w;
    });
}

std::size_t line_count() const { return _lines->size(); }

const std::string& line(std::size_t i) const {
```

```
    return (*_lines)[i];
  }

  // Two documents share data iff they are identical.
  bool shares_data_with(const document& other) const {
    return _lines.identical_to(other._lines);
  }
};
```

The `insert` and `erase` operations demonstrate the two-argument `modify` overload: when the document's lines are exclusively owned, the in-place action is used; when the lines are shared, the constructive transform builds a new vector from scratch, avoiding a redundant copy followed by a mutation.

The `shares_data_with` method demonstrates `identical_to`. Two documents that have not yet diverged will return `true`, allowing an editor to skip redundant display refreshes or diffs.