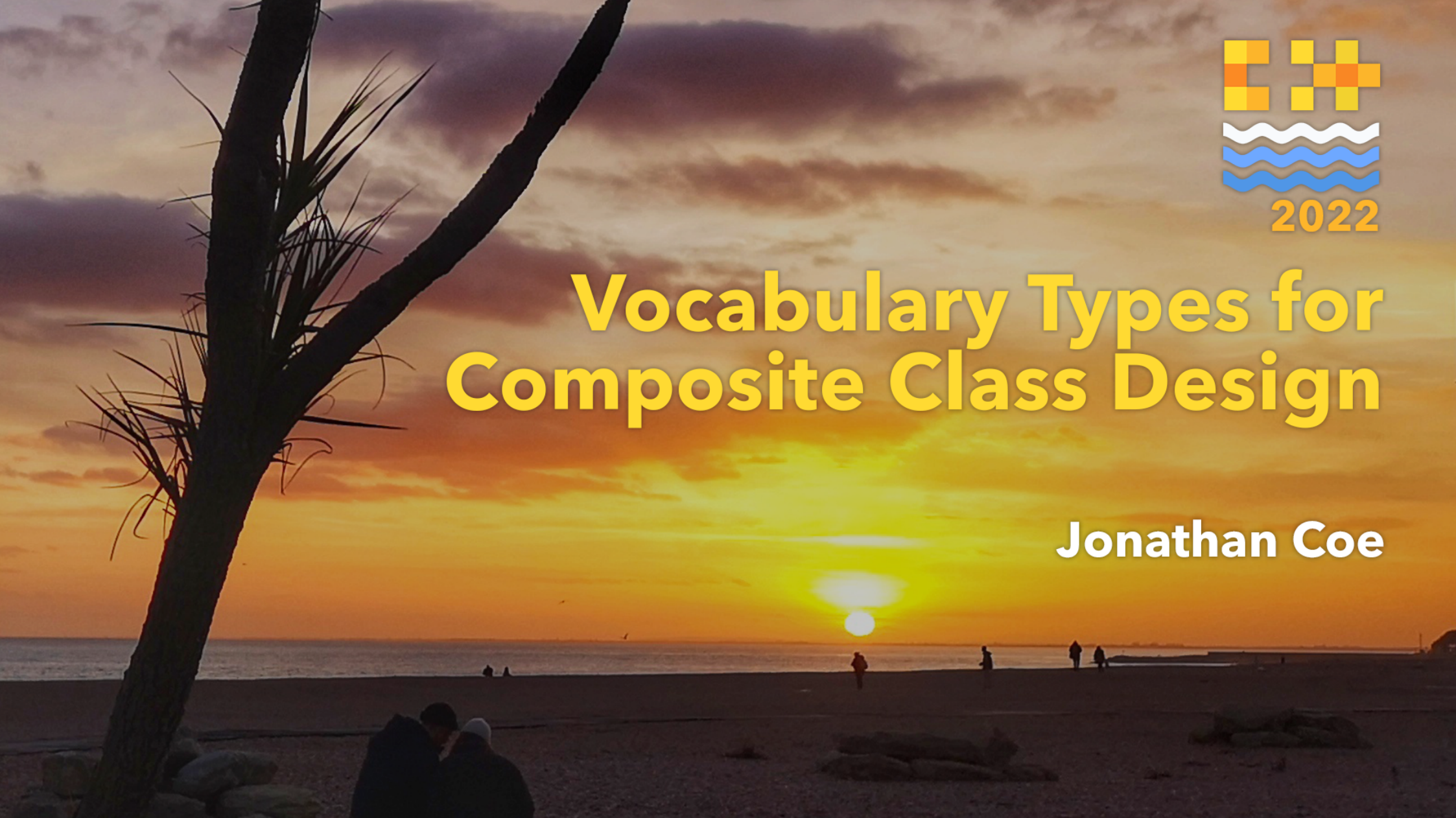
2022
Vocabulary Types for Composite Class Design
Jonathan Coe

Jonathan B. Coe & Antony Peacock

https://github.com/jbcoe/indirect_value

https://github.com/jbcoe/polymorphic_value

- We recommend the use of two new class templates, `indirect_value<T>` and `polymorphic_value<T>` to make the design of composite classes simple and correct.
- We'll look into some of the challenges of composite class design and see which problems are unsolved by current vocabulary types.
- We'll look at implementations of the two proposed new types.

# Encapsulation with classes (and structs)

Classes (and structs) let us group together logically associated data and functions that operate on that data:

```
class Circle {
    std::pair<double, double> position_;
    double radius_;
    std::string colour_;

public:
    std::string_view colour() const;
    double area() const;
};
```

# Encapsulation with classes (and structs) II

User-defined types can be used for member data:

```
struct Point {
    double x, y;
};

class Circle {
    Point position_;
    double radius_;
    std::string colour_;
};
```

# Special member functions

We can define special member functions to create, copy, move or destroy instances of our class:

```
class Circle {
    Circle(std::string_view colour, double radius, Point position);

    Circle(const Circle&);
    Circle& operator=(const Circle&);

    Circle(Circle&&);
    Circle& operator=(Circle&&);

    ~Circle();
};
```

# Compiler generated functions

- Under certain conditions, the compiler can generate special member functions for us.
- Members of the class will be copied/moved/deleted in turn by compiler-generated functions.
- We can use the amazing Compiler Explorer to see an example.

# Compiler generated functions II

```
class A {
  public:
    A();
    A(const A&);
    ~A();
};

class B {
    A a_;
  public:
    int foo();
};

int do_something(B b) {
    B b2(b);
    return b2.foo();
}
```

# Compiler generated functions III

X86 assembly generated from https://godbolt.org/ with `-O3 -fno-exceptions`

```
do_something(B):
        push    rbx
        mov     rsi, rdi
        sub     rsp, 16
        lea     rdi, [rsp+15]
        call    A::A(A const&) [complete object constructor]
        lea     rdi, [rsp+15]
        call    B::foo()
        lea     rdi, [rsp+15]
        mov     ebx, eax
        call    A::~A() [complete object destructor]
        add     rsp, 16
        mov     eax, ebx
        pop     rbx
        ret
```

# Compiler generated functions IV

- We can specify special member functions for classes we define.
- The compiler will (sometimes) generate special member functions for us.
- Generated special member functions are member-by-member calls to the appropriate special member function of each member object.

# Compiler generated functions with pointer members

When a class contains pointer members, the compiler generated special member functions will copy/move/delete the pointer but not the pointee:

```
struct A {
    A(...);
    B* b;
};

A a(...);
A a2(a);
assert(a.b == a2.b);
```

This might require us to write our own versions of the special member functions, something we'd sorely like to avoid having to do.

# `const` in C++

Member functions in C++ can be const-qualified:

```
struct A {
    void foo() const;
    void foo();
    void bar();
};
```

When an object is accessed through a const-access-path then only const-qualified member functions can be called:

```
const A a;
a.bar();
```

```
error: passing 'const A' as 'this' argument discards qualifiers
```

# `const` in C++ II

We get a const-access-path by directly accessing a const-qualified object or accessing an object through a reference or pointer to a const-qualified object.

Pointers can be const-qualifed and can point to const-qualified objects

```
A*;               // non-const pointer to a non-const A.
A* const;         // const pointer to a non-const A.
const A*;         // non-const pointer to a const A.
const A* const;   // const pointer to a const A.
```

Note that references cannot be made to refer to a different object although they can refer to a const-qualified or non-const-qualified object.

# `const` propagation

An object's member data becomes const-qualified when the object is accessed through a const-access-path:

```
class A {
  public:
    void foo(); // non-const
};

class B {
    A a_;
  public:
    void bar() const { a_.foo(); }
};
```

```
error: passing 'const A' as 'this' argument discards qualifiers
```

# `const` propagation and reference types

- Pointer (or reference) member data becomes const-qualified when accessed through a const-access-path, but the const-ness does not propagate through to the pointee.
- Pointers can't be made to point at different objects when accessed through a const-access-path but the object they point to can be accessed in a non-const manner.
- const-propagation must be borne in mind when designing composite classes for const-correctness.

# class-instance members

Class-instance members are often a good option for member data.

```
class A {
  B b_;
  C c_;
};
```

Class-instance members ensure const-correctness.

Compiler generated special member functions will be correct.

# Repeated member data

We might have cause to store a variable number of objects as part of our class

```
class Talk {
    Person speaker_;
    std::vector<Person> audience_;
}
```

Standard library containers like `vector` and `map` support a variable number of objects and have special member functions that will handle the contents of the container.

# Indirect storage

We may have member data that is too big to be sensibly stored directly in the class.

If member data is accessed infrequently we might want it stored elsewhere to keep cache lines hot.

```
class A {
    Data data_;
    BigData big_data_; // We want this stored elsewhere.
}
```

# Incomplete types

If the definition of a member is not available when a class is defined then we'll need to store the member as an incomplete type.

This can come about it node-like structures:

```
class Node {
    int data_;
    Node next_; // won't compile as `Node` is not yet defined.
}
```

# The Pointer To Implementation Pattern

The PIMPL pattern can be used to reduce compile times and keep ABI stable.
We store an incomplete type which defines the implementation detail of our class.

This can come about it node-like structures:

```
class A {
  public:
    int foo();
    double bar();
  private:
    Impl implementation_; // won't compile as `Impl` is not yet defined.
}
```

Where `A` is defined in a header file we want to define `Impl` in the associated `cc` source file.

# Polymorphism

We might require a member of our composite class to be one of a number of types.

- A Zoo could contain a list of Animals of different types.
- A code_checker could contain different checking_tools.
- A Game could contain different GameEntities.
- A Portfolio could contain different kinds of FinancialInstrument.

Our class will need to reserve storage for our polymorphic data member.

# Closed-set polymorphism

Closed-set polymorphism gives users of a class a choices for member data from a known set of types. We can use sum-types like `optional` and `variant` to represent this.

```
class Taco {
    std::optional<Avocado> avocado_;
    std::variant<Chicken, Pork, Beef> meat_;
    std::variant<Chipotle, GhostPepper, Hot> sauce_;
};
```

Storing a closed-set polymorphic member directly is possible as `variant` and `optional` reserve enough memory for the largest possible type.

# Open-set polymorphism

Open-set polymorphism allows users of a class represent member data with types that were not known when the class was defined.

```
struct SimulationObject {
    virtual ~SimulationObject();
    virtual void update() = 0;
};

class Simulation {
    ???? simulation_objects_;
};
```

Storing open-set polymorphic objects is challenging. We've no idea how much memory any of the objects might take so direct storage of the data is not possible.

# pointer (and reference) members

We can support polymorphism and incomplete types by storing a pointer as a member.

The pointer can be an incomplete type:

```
class A {
    class B* b_;
    class A* next_;
}
```

or the base type in a class heirarchy:

```
struct Shape {
    virtual void Draw() const = 0;
};

class Picture {
    Shape* shape_;
}
```

# Collections of pointers as members

We can store multiple pointers to objects in our class in standard library collections:

```
struct Animal {
    const char* MakeNoise() const = 0;
};

class Zoo {
    std::vector<Animal*> animals_;
};

class SafeZoo {
    std::map<std::string, std::vector<Animal*>> animals_;
};
```

# Issues with pointer members

- Compiler generated special members functions handle only the pointer, not the pointee.
- `const` will not propagate to pointees
- If we want to model ownership in our composite then we'll have to do work:
  - Manually maintain special member functions.
  - Check const-qualified member functions for const correctness.

# Improving on pointer members

C++'s handling of pointers is not wrong, but in the examples above, we've failed to communicate what we mean to the compiler.

There are instances where pointer members perfectly model what we want to express but such instances are not composites:

```
class Worker {
    std::string name_;
    Manager* manager_;
};
```

Let's see if we can do better.

# Smart pointers

Smart pointers have different semantics to raw pointers and can be used to express intent to the compiler.

- Ownership can be transferred.
- Resources can be freed on destruction.

C++98 had std::auto_ptr. With the introduction of move semantics, we have improved smart pointers in C++11.

[C++11 deprecated std::auto_ptr. C++17 removed it.]

# `std::unique_ptr<T>` members

```
class A {
    std::unique_ptr<B> b_;
};
```

This is an improvement over raw-pointer members.

We now have a correct compiler-generated destructor, move-constructor and move-assignment operator.

`std::unique_ptr` is non-copyable, so the compiler won't generate a copy constructor or assignement operator for us.

`std::unique_ptr` does not propagate `const` so we'll have to check our implementations of const-qualified member functions ourselves.

# `std::shared_ptr<T>` members

```
class A {
    std::shared_ptr<B> b_;
};
```

With `shared_ptr` members, the compiler can generate all special member functions for us.

Sadly this is not much of an improvement as the copy constructor and assignment operator will not copy the `B` object that we point to, only add references to it.

We now have mutable shared state and still no const-propagation.

# `std::shared_ptr<const T>` members

```
class A {
    std::shared_ptr<const B> b_;
};
```

Again, the compiler can generate all special member functions for us.

Copy and assignement will add references to the same `B` object but seeing as it's immutable that could be ok (so long as it's not mutable by another route).

We've lost the ability to call any mutation method of the B object though as any access to it is through a const-access-path.

# New proposed classes

We propose the addition of two new class templates

- `polymorphic_value`
- `indirect_value`

# `polymorphic_value<T>` members

```
class A {
    polymorphic_value<B> b_; // proposed addition to the standard
};
```

A `polymorphic_value` member allows the compiler to generate special member functions correctly and propagates `const` so that const-qualified member functions can be verified by the compiler.

`B` is allowed to be a base class and `b_` can store an instance of a derived type. Copying and deleting derived types works correctly as polymorphic_value is implemented using type-erasure.

# Design of `polymorphic_value`

`polymorphic_value` is a class template taking a single template argument - the base class of the types that we want to store.

```
template <class T>
class polymorphic_value;
```

## Constructors

```
polymorphic_value() noexcept;

template <class U, class... Ts> // restrictions apply
explicit polymorphic_value(std::in_place_type_t<U>, Ts&&... ts);

template <class U, class C=default_copy<U>, class D=default_delete<U>>
explicit polymorphic_value(U* p, C c=C{}, D d=D{});  // restrictions apply
```

## Move and copy

```
polymorphic_value(const polymorphic_value& p);
polymorphic_value(polymorphic_value&& p) noexcept;

template <class U> // restrictions apply
polymorphic_value(const polymorphic_value<U>& p);

template <class U> // restrictions apply
polymorphic_value(polymorphic_value<U>&& p);
```

## Assignment

```
polymorphic_value& operator=(const polymorphic_value& p);

polymorphic_value& operator=(polymorphic_value &&p) noexcept;
```

## Modifiers and observers

```
void swap(polymorphic_value<T>& p) noexcept;

explicit operator bool() const noexcept;

T& operator*();
T* operator->();

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

## Creation

```
template <class T, class ...Ts>
polymorphic_value<T> make_polymorphic_value(Ts&& ...ts);
```

# Implementing `polymorphic_value<T>`

```
template <class T> class polymorphic_value {
  std::unique_ptr<control_block<T>> cb_;
  T* ptr_ = nullptr;

 public:
  polymorphic_value() = default;

  polymorphic_value(const polymorphic_value& p) : cb_(p.cb_->clone()) {
    ptr_ = cb_->ptr();
  }

  T* operator->() { return ptr_; }
  const T* operator->() const { return ptr_; }

  T& operator*() { return *ptr_; }
  const T& operator*() const { return *ptr_; }
};
```

All of the real work is done by the control block.

# Implementing the control block

Control blocks will inherit from a base-class:

```
template <class T>
struct control_block
{
  virtual ~control_block() = default;
  virtual T* ptr() = 0;
  virtual std::unique_ptr<control_block> clone() const = 0;
};
```

## Construction from a value

We can support constructors of the form:

```
template<class U, class ...Ts> // restrictions apply
polymorphic_value(std::in_place_type<U>, Ts...ts);
```

with a suitable control block:

```
template <class T, class U>
class direct_control_block : public control_block<T> {
  U u_;
 public:
  template <class... Ts>
  explicit direct_control_block(Ts&&... ts) :
    u_(U(std::forward<Ts>(ts)...)) {}

  T* ptr() override { return &u_; }

  std::unique_ptr<control_block<T>> clone() const override {
    return std::make_unique<direct_control_block>(*this);
  }
};
```

The control block knows how to copy and delete the object it owns.

```
template<class U,
         class = std::enable_if_t<
           std::is_convertible<U*, T*>::value>>
polymorphic_value(const U& u) :
  cb_(std::make_unique<direct_control_block<T,U>>(u))
{
  ptr_ = cb_->ptr();
}
```

## Copying from another polymorphic value

We can support constructors of the form:

```
template <class U> // restrictions apply
polymorphic_value(const polymorphic_value<U>& p);
```

with a suitable control block:

```
template <class T, class U>
class delegating_control_block : public control_block<T> {
  std::unique_ptr<control_block<U>> delegate_;

public:
  explicit delegating_control_block(
    std::unique_ptr<control_block<U>> b) :
      delegate_(std::move(b)) {}

  std::unique_ptr<control_block<T>> clone() const override {
    return std::make_unique<delegating_control_block>(
      delegate_->clone());
  }

  T* ptr() override {
    return delegate_->ptr();
  }
};
```

```
template <class U,
          class = std::enable_if_t<
            std::is_convertible<U*, T*>::value>>
polymorphic_value(const polymorphic_value<U>& p)
{
  polymorphic_value<U> tmp(p);

  ptr_ = tmp.ptr_;
  cb_ = std::make_unique<delegating_control_block<T, U>>(
    std::move(tmp.cb_));
}
```

## Construction from a pointer

We can support constructors of the form:

```
template <class U,
          class C = default_copy<U>,
          class D = default_delete<U>> // restrictions apply
explicit polymorphic_value(U* u,
                           C copier = C{},
                           D deleter = D{});
```

with a suitable control block:

```
template <class U,
          class C = default_copy<U>,
          class D = default_delete<U>>
class pointer_control_block : public control_block<T> {
  std::unique_ptr<U, D> p_;
  C c_;

public:
  explicit pointer_control_block(U* u, C c = C{}, D d = D{})
      : c_(std::move(c)), p_(u, std::move(d)) {}

  std::unique_ptr<control_block<T>> clone() const override {
    assert(p_);
    return std::make_unique<pointer_control_block>(
      c_(*p_), c_, p_.get_deleter());
  }

  T* ptr() override {
    return p_.get();
  }
};
```

```
template <class U,
          class C = default_copy<U>,
          class D = default_delete<U>,
          class = std::enable_if_t<
            std::is_convertible<U*, T*>::value>>
explicit polymorphic_value(U* u,
                           C copier = C{},
                           D deleter = D{})
{
  if (!u) return;

  assert(typeid(*u) == typeid(U)); // Here be dragons!

  cb_ = std::make_unique<pointer_control_block<T, U, C, D>>(
      u, std::move(copier), std::move(deleter));
  ptr_ = u;
}
```

# Using `polymorphic_value<T>` in your code

`polymorphic_value` is a single-file-, header-only-library and can be included in your C++ project by using the header file from our reference implementation

https://github.com/jbcoe/polymorphic_value

```
jbcoe/polymorphic_value is licensed under the

MIT License

A short and simple permissive license with conditions only
requiring preservation of copyright and license notices.
Licensed works, modifications, and larger works may be
distributed under different terms and without source code.
```

# `indirect_value<T>` members

```
class A {
    indirect_value<B> b_; // proposed addition to the standard
};
```

An `indirect_value` member allows the compiler to generate special member functions correctly and propagates `const` so that const-qualified member functions can be verified by the compiler.

`B` can be an incomplete type. `b_` can only store an instance of `B`. Copying and deleting the owned object works without virtual dispatch.

# Design of `indirect_value`

`indirect_value` is a class template taking three template argument - the type we want to store, a copier and a deleter.

Only the first template argument is mandatory, `default_copy` and `default_delete` will be used if later template arguments are not supplied.

```
template <class T, class C=std::default_copy<T>, class D=std::default_delete<D>>
class indirect_value;
```

## Constructors

```
indirect_value() noexcept;

template <class U, class... Ts>
explicit indirect_value(std::in_place_t, Ts&&... ts);

template <class U, class C=default_copy<U>, class D=default_delete<U>>
explicit indirect_value(U* p, C c=C{}, D d=D{});  // restrictions apply
```

## Move and Copy

```
indirect_value(const indirect_value& i);
indirect_value(indirect_value&& i) noexcept;
```

## Assignment

```
indirect_value& operator=(const indirect_value& i);
indirect_value& operator=(indirect_value&& i) noexcept;
```

## Modifiers and observers

```
void swap(indirect_value<T>& p) noexcept;

explicit operator bool() const noexcept;

T& operator*();
T* operator->();

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

## Three way comparison and hash

`indirect_value` specializes `std::hash` when the stored type specializes `std::hash`.

`indirect_value` supports binary comparsion operations in cases where the stored type supports binary comparsion operations.

`indirect_value` supports three-way-comparison (operator `<=>`) in cases where the stored type supports three-way-comparison.

As `indirect_value` can be empty it supports hash and comparison operations in the same way as `std::optional`.

# Implementing `indirect_value<T>`

`indirect_value` could be naively implemented with a raw pointer:

```
template <class T, class C = default_copy<T>, class D = std::default_delete<T>>
class indirect_value {
    T* ptr_;
    C c_;
    D d_;
  public:
    // Constructors elided

    T* operator->() noexcept { return ptr_; }
    const T* operator->() const noexcept { return ptr_; }
    T& operator*() & noexcept { return *ptr_; }
    const T& operator*() const& noexcept { return *ptr_; }
};
```

## Copying `indirect_value<T>`

`indirect_value<T>` does not need type erasure or virtual dispatch to create copies. The type of the owned object is know at compile time - if present, it must be a `T`.

If an object is present, the copy constructor of of `indirect_value<T>` calls `c_(*ptr_)` to get a new object.

## Deleting `indirect_value<T>`

`indirect_value<T>` does not need type erasure or virtual dispatch to create copies. The type of the owned object is know at compile time - if present, it must be a `T`.

The destructor of of `indirect_value<T>` calls `d_(ptr_)` to destroy an owned object.

## Avoiding unnecessary storage

We'd like to avoid allocating storage for the copier and deleter where possible.

If the objects have no member data then they don't need to reserve space.

Our reference implementation uses the empty base class optimisation to eliminate storage for empty copiers and deleters:

```
template <class T, class C = default_copy<T>, class D = std::default_delete<T>>
class ISOCPP_P1950_EMPTY_BASES indirect_value
    : private indirect_value_copy_base<C>,
      private indirect_value_delete_base<D> {
```

# Avoiding unnecessary storage II

One could employ `[[no_unique_address]]` from C++20 to avoid allocating storage for empty copiers and deleters.

```
template <class T, class C = default_copy<T>, class D = std::default_delete<T>>
class indirect_value {
    T* ptr_;
    [[no_unique_address]] C c_;
    [[no_unique_address]] D d_;
     ...
};
```

# Using `indirect_value<T>` in your code

`indirect_value` is a single-file-, header-only-library and can be included in your C++ project by using the header file from our reference implementation

https://github.com/jbcoe/indirect_value

```
jbcoe/indirect_value is licensed under the

MIT License

A short and simple permissive license with conditions only
requiring preservation of copyright and license notices.
Licensed works, modifications, and larger works may be
distributed under different terms and without source code.
```

# Values not references

The two class templates we've proposed are value types and treat the objects they own as values:

- They delete their owned objects upon destruction.
- They copy their owned objects (correctly) when copied.

Value types are the right choice for the design of composite classes (unless we want to do a bunch of extra work).

Both classes can be default constructed in an empty state, this makes them regular types and means that they will interact well with existing collections like `std::vector` and `std::map`.

# Standardisation efforts

There are two papers in flight to add `polymorphic_value` and `indirect_value` to a future version of the C++ standard.

- Polymorphic Value https://wg21.link/p0201r5
- Indirect Value https://wg21.link/p1950r0

Now that travel restrictions are lifted, we hope to resume our work on standardisation.

# Acknowledgements

Many thanks to Sean Parent and to the Library Evolution Working group from the C++ Standards committee for interesting early discussion in the design of `polymorphic_value`.

Thanks to our GitHub contributors who've made worked with us to improve our reference implementations.