A polymorphic value-type for C++

ISO/IEC JTC1 SC22 WG21 Programming Language C++

P0201R4

Working Group: Library Evolution, Library

Date: 2018-10-05

Jonathan Coe <jonathanbcoe@gmail.com>

Sean Parent <sparent@adobe.com>

Change history

Changes in P0201R4

- Clarify authors' agreement with LEWG design changes.
- Add wording to clarify meaning of custom copier and deleter.
- Make constructors explicit and remove converting assignment.
- Add a second template parameter to `make_polymorphic_value` to facilitate construction of objects of derived classes.

Changes in P0201R3

- Add rationale for absence of allocator support.

Changes in P0201R2

- Change name to `polymorphic_value`.
- Remove `operator <<`.
- Add construction and assignment from values.
- Use `std::default_delete`.
- Rename `std::default_copier` to `std::default_copy`.
- Add notes on empty state and pointer constructor.
- Add `bad_polymorphic_value_construction` exception when static and dynamic type of pointee mismatch and no custom copier or deleter are supplied.
- Add clarifying note to say that a small object optimisation is allowed.

Changes in P0201R1

- Change name to `indirect`.

- Remove `static_cast`, `dynamic_cast` and `const_cast` as `polymorphic_value` is modelled on a value not a pointer.
- Add `const` accessors which return `const` references/pointers.
- Remove pointer-accessor `get`.
- Remove specialization of `propagate_const`.
- Amended authorship and acknowledgements.
- Added support for custom copiers and custom deleters.
- Removed hash and comparison operators.

TL;DR

Add a class template, `polymorphic_value<T>`, to the standard library to support polymorphic objects with value-like semantics.

Introduction

The class template, `polymorphic_value`, confers value-like semantics on a free-store allocated object. A `polymorphic_value<T>` may hold an object of a class publicly derived from T, and copying the `polymorphic_value<T>` will copy the object of the derived type.

Motivation: Composite objects

Use of components in the design of object-oriented class hierarchies can aid modular design as components can be potentially re-used as building-blocks for other composite classes.

We can write a simple composite object formed from two components as follows:

```
// Simple composite
class CompositeObject_1 {
  Component1 c1_;
  Component2 c2_;

  public:
  CompositeObject_1(const Component1& c1,
                    const Component2& c2) :
                    c1_(c1), c2_(c2) {}

  void foo() { c1_.foo(); }
  void bar() { c2_.bar(); }
};
```

The composite object can be made more flexible by storing pointers to objects allowing it to take derived components in its constructor. (We store pointers to the components rather than

references so that we can take ownership of them).

```
// Non-copyable composite with polymorphic components (BAD)
class CompositeObject_2 {
  IComponent1* c1_;
  IComponent2* c2_;

  public:
  CompositeObject_2(const IComponent1* c1,
                    const IComponent2* c2) :
                    c1_(c1), c2_(c2) {}

  void foo() { c1_->foo(); }
  void bar() { c2_->bar(); }

  CompositeObject_2(const CompositeObject_2&) = delete;
  CompositeObject_2& operator=(const CompositeObject_2&) = delete;

  CompositeObject_2(CompositeObject_2&& o) : c1_(o.c1_), c2_(o.c2_) {
    o.c1_ = nullptr;
    o.c2_ = nullptr;
  }

  CompositeObject_2& operator=(CompositeObject_2&& o) {
    delete c1_;
    delete c2_;
    c1_ = o.c1_;
    c2_ = o.c2_;
    o.c1_ = nullptr;
    o.c2_ = nullptr;
  }

  ~CompositeObject_2()
  {
    delete c1_;
    delete c2_;
  }
};
```

`CompositeObject_2`'s constructor API is unclear without knowing that the class takes ownership of the objects. We are forced to explicitly suppress the compiler-generated copy constructor and copy assignment operator to avoid double-deletion of the components `c1_` and `c2_`. We also need to write a move constructor and move assignment operator.

Using `unique_ptr` makes ownership clear and saves us writing or deleting compiler generated methods:

```
// Non-copyable composite with polymorphic components
class CompositeObject_3 {
  std::unique_ptr<IComponent1> c1_;
  std::unique_ptr<IComponent2> c2_;

  public:
  CompositeObject_3(std::unique_ptr<IComponent1> c1,
                    std::unique_ptr<IComponent2> c2) :
                    c1_(std::move(c1)), c2_(std::move(c2)) {}

  void foo() { c1_->foo(); }
  void bar() { c2_->bar(); }
};
```

The design of `CompositeObject_3` is good unless we want to copy the object.

We can avoid having to define our own copy constructor by using shared pointers. As `shared-ptr`'s copy constructor is shallow, we need to modify the component pointers to be pointers-to `const` to avoid introducing shared mutable state [S.Parent].

```
// Copyable composite with immutable polymorphic components class
class CompositeObject_4 {
  std::shared_ptr<const IComponent1> c1_;
  std::shared_ptr<const IComponent2> c2_;

  public:
  CompositeObject_4(std::shared_ptr<const IComponent1> c1,
                    std::shared_ptr<const IComponent2> c2) :
                    c1_(std::move(c1)), c2_(std::move(c2)) {}

  void foo() { c1_->foo(); }
  void bar() { c2_->bar(); }
};
```

`CompositeObject_4` has polymorphism and compiler-generated destructor, copy, move and assignment operators. As long as the components are not mutated, this design is good. If non-const methods of components are used then this won't compile.

Using `polymorphic_value` a copyable composite object with polymorphic components can be written as:

```
// Copyable composite with mutable polymorphic components
class CompositeObject_5 {
```

```
  std::polymorphic_value<IComponent1> c1_;
  std::polymorphic_value<IComponent2> c2_;

  public:
  CompositeObject_5(std::polymorphic_value<IComponent1> c1,
                    std::polymorphic_value<IComponent2> c2) :
                    c1_(std::move(c1)), c2_(std::move(c2)) {}

  void foo() { c1_->foo(); }
  void bar() { c2_->bar(); }
};
```

The component `c1_` can be constructed from an instance of any class that inherits from `IComponent1`. Similarly, `c2_` can be constructed from an instance of any class that inherits from `IComponent2`.

`CompositeObject_5` has a compiler-generated destructor, copy constructor, move constructor, assignment operator and move assignment operator. All of these compiler-generated functions will behave correctly.

Deep copies

To allow correct copying of polymorphic objects, `polymorphic_value` uses the copy constructor of the owned derived-type object when copying a base type `polymorphic_value`. Similarly, to allow correct destruction of polymorphic component objects, `polymorphic_value` uses the destructor of the owned derived-type object in the destructor of a base type `polymorphic_value`.

The requirements of deep-copying can be illustrated by some simple test code:

```
// GIVEN base and derived classes.
class Base { virtual void foo() const = 0; };
class Derived : public Base { void foo() const override {} };

// WHEN a polymorphic_value to base is formed from a derived object
polymorphic_value<Base> poly(Derived());
// AND the polymorphic_value to base is copied.
auto poly_copy = poly;

// THEN the copy owns a distinct object
assert(&*poly != &*poly_copy);
// AND the copy owns a derived type.
assert(dynamic_cast<Derived*>(*&poly_copy));
```

Note that while deep-destruction of a derived class object from a base class pointer can be performed with a virtual destructor, the same is not true for deep-copying. C++ has no concept of a virtual copy constructor and we are not proposing its addition. The class template `shared_ptr`

already implements deep-destruction without needing virtual destructors; deep-destruction and deep-copying can be implemented using type-erasure [Impl].

Pointer constructor

`polymorphic_value` can be constructed from a pointer and optionally a copier and/or deleter. The `polymorphic_value` constructed in this manner takes ownership of the pointer. This constructor is potentially dangerous as a mismatch in the dynamic and static type of the pointer will result in incorrectly synthesized copiers and deleters, potentially resulting in slicing when copying and incomplete deletion during destruction.

```
class Base { /* methods and members */ };
class Derived : public Base { /* methods and members */ };

Derived d = new Derived();
Base* p = d; // static type and dynamic type differ
polymorphic_value<Base> poly(p);

// This copy will have been made using Base's copy constructor.
polymorphic_value<Base> poly_copy = poly;

// Destruction of poly and poly_copy uses Base's destructor.
```

While this is potentially error prone, we have elected to trust users with the tools they are given. `shared_ptr` and `unique_ptr` have similar constructors and issues. There are more constructors for `polymorphic_value` of a less expert-friendly nature that do not present such dangers including a factory method `make_polymorphic_value`.

Static analysis tools can be written to find cases where static and dynamic types for pointers passed in to `polymorphic_value` constructors are not provably identical.

If the user has not supplied a custom copier or deleter, an exception `bad_polymorphic_value_construction` is thrown from the pointer-constructor if the dynamic and static types of the pointer argument do not agree. In cases where the user has supplied a custom copier or deleter it is assumed that they will do so to avoid slicing and incomplete destruction: a class heirarchy with a custom `Clone` method and virtual destructor would make use of `Clone` in a user-supplied copier.

Empty state

`polymorphic_value` presents an empty state as it is desirable for it to be cheaply constructed and then later assigned. In addition, it may not be possible to construct the `T` of a `polymorphic_value<T>` if it is an abstract class (a common intended use pattern). While permitting an empty state will necessitate occasional checks for `null`, `polymorphic_value` is intended to replace uses of pointers or smart pointers where such checks are also necessary. The benefits of default constructability (use in vectors and maps) outweigh the costs of a possible empty state.

Lack of hashing and comparisons

For a given user-defined type, `T`, there are multiple strategies to make `polymorphic_value<T>` hashable and comparable. Without requiring additional named member functions on the type, `T`, or mandating that `T` has virtual functions and RTTI, the authors do not see how `polymorphic_value` can generically support hashing or comparisons. Incurring a cost for functionality that is not required goes against the 'pay for what you use' philosophy of `C++`.

For a given user-defined type `T` the user is free to specialize `std::hash` and implement comparison operators for `polymorphic_value<T>`.

Custom copiers and deleters

The resource management performed by `polymorphic_value` - copying and destruction of the managed object - can be customized by supplying a *copier* and *deleter*. If no copier or deleter is supplied then a default copier or deleter will be used.

The default deleter is already defined by the standard library and used by `unique_ptr`.

We define the default copier in technical specifications below.

Allocator Support

The design of `polymorphic_value` is similar to that of `std::any`, which does not have support for allocators.

`polymorphic_value`, like `std::any` and `std::function`, is implemented in terms of type-erasure. There are technical issues with storing an allocator in a type-erased context and recovering it later for allocations needed during copy assignment [P0302r1].

Until such technical obstacles can be overcome, `polymorphic_value` will follow the design of `std::any` and `std::function` (post C++17) and will not support allocators.

Design changes from `cloned_ptr`

The design of `polymorphic_value` is based upon `cloned_ptr` (from an early revision of this paper) and modified following advice from LEWG. The authors (who unreservedly agree with the design direction suggested by LEWG) would like to make explicit the cost of these design changes.

`polymorphic_value<T>` has value-like semantics: copies are deep and `const` is propagated to the owned object. The first revision of this paper presented `cloned_ptr<T>` which had mixed pointer/value semantics: copies are deep but `const` is not propagated to the owned object. `polymorphic_value` can be built from `cloned_ptr` and `propagate_const` but there is no way to remove `const` propagation from `polymorphic_value`.

As `polymorphic_value` is a value, `dynamic_pointer_cast`, `static_pointer_cast` and `const_pointer_cast` are not provided. If a `polymorphic_value` is constructed with a custom copier or deleter, then there is no way for a user to implement cast operations like those that are provided by the standard for `std::shared_ptr`.

No implicit conversions

Following design feedback, `polymorphic_value`'s constructors have been made explicit so that surprising implicit conversions cannot take place. Any conversion to a `polymorphic_value` must be explicitly requested by user-code.

The converting assignment operators that were present in earlier drafts have also been removed.

For a base class, `BaseClass`, and derived class, `DerivedClass`, the converting assignment

```
polymorphic_value<DerivedClass> derived;
polymorphic_value<Base> base = derived;
```

is no longer valid, the conversion must be made explicit:

```
polymorphic_value<DerivedClass> derived;
auto base = polymorphic_value<Base>(derived);
```

The removal of converting assigments makes `make_polymorphic_value` slightly more verbose to use:

```
polymorphic_value<Base> base = make_polymorphic_value<DerivedClass>(args);
```

is not longer valid and must be written as

```
auto base = polymorphic_value<Base>(make_polymorphic_value<DerivedClass>
(args));
```

This is somewhat cumbersome so `make_polymorphic_value` has been modified to take an optional extra template argument allowing users to write

```
polymorphic_value<Base> base = make_polymorphic_value<Base, DerivedClass>
(args);
```

The change from implicit to explicit construction is deliberately conservative. One can change explicit constructors into implicit constructors without breaking code (other than SFINAE checks), the reverse is not true. Similarly, converting assignments could be added non-disruptively but not so readily removed.

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

X.X Class template `default_copy` [default.copy]

```
namespace std {
template <class T> struct default_copy {
  T* operator()(const T& t) const;
};

} // namespace std
```

The class template `default_copy` serves as the default copier for the class template `polymorphic_value`.

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

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

- *Returns*: `new T(t)`.

X.Y Class `bad_polymorphic_value_construction` [bad_polymorphic_value_construction]

```
namespace std {
class bad_polymorphic_value_construction : public exception
{
  public:
    bad_polymorphic_value_construction() noexcept;

    const char* what() const noexcept override;
};
}
```

Objects of type `bad_polymorphic_value_construction` are thrown to report invalid construction of a `polymorphic_value` from a pointer argument.

```
bad_polymorphic_value_construction() noexcept;
```

- *Effects*: Constructs a `bad_polymorphic_value_construction` object.

```
const char* what() const noexcept override;
```

- *Returns*: An implementation-defined NTBS.

X.Z Class template `polymorphic_value` [polymorphic_value]

X.Z.1 Class template `polymorphic_value` general [polymorphic_value.general]

The `polymorphic_value` class template stores a pointer to another object. The object pointed to by the pointer is referred to as an owned object. `polymorphic_value` implements value semantics: the owned object is copied or destroyed when the `polymorphic_value` is copied or destroyed.

A `polymorphic_value`, `v`, will dispose of its owned object when `v` is itself destroyed (e.g., when leaving block scope (9.7)).

A `polymorphic_value` object is empty if there is no owned object (the stored pointer is `nullptr`).

Copying a non-empty `polymorphic_value` will copy the owned object so that the copied `polymorphic_value` will have its own unique copy of the owned object.

Copying from an empty `polymorphic_value` produces another empty `polymorphic_value`.

Copying and disposal of the owned object can be customized by supplying a copier and deleter.

If a `polymorphic_value` is constructed from a pointer then it is said to have a custom copier and deleter. Any `polymorphic_value` instance constructed from another `polymorphic_value` instance constructed with a custom copier and deleter will also have a custom copier and deleter.

The template parameter `T` of `polymorphic_value` shall be a non-union class type. Otherwise the program is ill-formed.

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

[Note: Implementations are encouraged to avoid the use of dynamic memory for ownership of small objects.]

X.Z.2 Class template `polymorphic_value` synopsis [polymorphic_value.synopsis]

```
namespace std {
template <class T> class polymorphic_value {
 public:
  using element_type = T;

  // Constructors
  constexpr polymorphic_value() noexcept;

  constexpr polymorphic_value(nullptr_t) noexcept;

  template <class U> explicit polymorphic_value(U&& u);
```

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

  polymorphic_value(const polymorphic_value& p);
  template <class U>
    explicit polymorphic_value(const polymorphic_value<U>& p);

  polymorphic_value(polymorphic_value&& p) noexcept;
  template <class U>
    explicit polymorphic_value(polymorphic_value<U>&& p);

  // Destructor
  ~polymorphic_value();

  // Assignment
  polymorphic_value& operator=(const polymorphic_value& p);
  polymorphic_value& operator=(polymorphic_value&& p) noexcept;

  // Modifiers
  void swap(polymorphic_value& p) noexcept;

  // Observers
  T& operator*();
  T* operator->();
  const T& operator*() const;
  const T* operator->() const;
  explicit operator bool() const noexcept;
};

// polymorphic_value creation
template <class T, class U=T, class... Ts> polymorphic_value<T>
  make_polymorphic_value(Ts&&... ts);

// polymorphic_value specialized algorithms
template<class T>
  void swap(polymorphic_value<T>& p, polymorphic_value<T>& u) noexcept;

} // end namespace std
```

X.Z.3 Class template `polymorphic_value` constructors [polymorphic_value.ctor]

```
constexpr polymorphic_value() noexcept;
constexpr polymorphic_value(nullptr_t) noexcept;
```

- *Effects*: Constructs an empty `polymorphic_value`.

```
template <class U> explicit polymorphic_value(U&& u);
```

- *Remarks*: Let `V` be `remove_cvref_t<U>`. This constructor shall not participate in overload resolution unless `V*` is convertible to `T*`.
- *Effects*: Constructs a `polymorphic_value` whose owned object is initialised with `V(std::forward<U>(u))`.
- *Throws*: Any exception thrown by the selected constructor of `V` or `bad_alloc` if required storage cannot be obtained.

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

- *Remarks*: This constructor shall not participate in overload resolution unless `U*` is convertible to `T*`. A custom copier and deleter are said to be 'present' in a `polymorphic_value` initialized with this constructor.
- *Effects*: If `p` is null, creates an empty object, otherwise creates a `polymorphic_value` object that owns the pointer `p`.

  If `p` is non-null then the copier and deleter of the `polymorphic_value` constructed are initialized from `std::move(c)` and `std::move(d)`.
- *Requires*: `C` and `D` satisfy the requirements of CopyConstructible. If `p` is non-null then the expression `c(*p)` returns an object of type `U*`. The expression `d(p)` is well formed, has well defined behavior, and does not throw exceptions. Either `U` and `T` must be the same type, or the dynamic and static type of `*p` must be the same.
- *Throws*: `bad_polymorphic_value_construction` if `is_same_v<C, default_copy<U>>`, `is_same_v<D, default_delete<U>>` and `typeid(*p)!=typeid(U)`; `bad_alloc` if required storage cannot be obtained.
- *Postconditions*: `bool(*this) == bool(p)`.

```
polymorphic_value(const polymorphic_value& p);
template <class U> explicit polymorphic_value(const polymorphic_value<U>&
p);
```

- *Remarks*: The second constructor shall not participate in overload resolution unless `U*` is convertible to `T*`.
- *Effects*: Creates a `polymorphic_value` object that owns a copy of the object managed by `p`. If `p` has a custom copier then the copy is created by the copier in `p`. Otherwise the copy is

created by copy construction of the owned object. If `p` has a custom copier and deleter then the custom copier and deleter of the `polymorphic_value` constructed are copied from those in `p`.

- *Throws*: Any exception thrown by the copier or `bad_alloc` if required storage cannot be obtained.
- *Postconditions*: `bool(*this) == bool(p)`.

```
polymorphic_value(polymorphic_value&& p) noexcept;
template <class U> explicit polymorphic_value(polymorphic_value<U>&& p)
noexcept;
```

- *Remarks*: The second constructor shall not participate in overload resolution unless `U*` is convertible to `T*`.
- *Effects*: Ownership of the resource managed by `p` is transferred to the constructed `polymorphic_value`. Potentially move constructs the owned object (if the dynamic type of the owned object is no-throw move-constructible). If `p` has a custom copier and deleter then the copier and deleter of the `polymorphic_value` constructed are the same as those in `p`.
- *Throws*: `bad_alloc` if required storage cannot be obtained.
- *Postconditions*: `*this` contains the old value of `p`. `p` is empty.

[Note: This constructor can allow an implementation to avoid the need for dynamic memory allocation.]

X.Z.4 Class template `polymorphic_value` destructor [polymorphic_value.dtor]

```
~polymorphic_value();
```

- *Effects*: If get() == nullptr there are no effects. If a custom deleter `d` is present then `d(p)` is called and the copier and deleter are destroyed. Otherwise the destructor of the managed object is called.

X.Z.5 Class template `polymorphic_value` assignment [polymorphic_value.assignment]

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

- *Effects*: `*this` owns a copy of the resource managed by `p`. If `p` has a custom copier and deleter then the copy is created by the copier in `p`, and the copier and deleter of `*this` are copied from those in `p`. Otherwise the resource managed by `*this` is initialised by the copy constructor of the resource managed by `p`.

- *Throws*: Any exception thrown by the copier or `bad_alloc` if required storage cannot be obtained.
- *Returns*: `*this`.
- *Postconditions*: `bool(*this) == bool(p)`.

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

- *Effects*: Ownership of the resource managed by `p` is transferred to `this`. Potentially move constructs the owned object (if the dynamic type of the owned object is no-throw move-constructible). If `p` has a custom copier and deleter then the copier and deleter of `*this` are the same as those in `p`.
- *Throws*: `bad_alloc` if required storage cannot be obtained.
- *Returns*: `*this`.
- *Postconditions*: `*this` contains the old value of `p`. `p` is empty.

[Note: move construction of an owned object may be used by an implementation to avoid the need for use of dynamic memory.]

X.Z.6 Class template `polymorphic_value` modifiers [polymorphic_value.modifiers]

```
void swap(polymorphic_value& p) noexcept;
```

- *Effects*: Exchanges the contents of `p` and `*this`.

X.Z.7 Class template `polymorphic_value` observers [polymorphic_value.observers]

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

- *Requires*: `bool(*this)`.
- *Returns*: A reference to the owned object.

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

- *Requires*: `bool(*this)`.
- *Returns*: A pointer to the owned object.

```
explicit operator bool() const noexcept;
```

- *Returns*: `false` if the `polymorphic_value` is empty, otherwise `true`.

X.Z.8 Class template `polymorphic_value` creation [polymorphic_value.creation]

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

- *Returns*: A `polymorphic_value<T>` owning an object initialised with `U(std::forward<Ts>(ts)...)`.

[Note: Implementations are encouraged to avoid multiple allocations.]

X.Z.9 Class template `polymorphic_value` specialized algorithms [polymorphic_value.spec]

```
template <typename T>
void swap(polymorphic_value<T>& p, polymorphic_value<T>& u) noexcept;
```

- *Effects*: Equivalent to `p.swap(u)`.

Acknowledgements

The authors would like to thank Maciej Bogus, Matthew Calbrese, Germán Diago, Louis Dionne, Bengt Gustafsson, Tom Hudson, Stephan T Lavavej, Tomasz Kamiński, David Krauss, Thomas Koeppe, LanguageLawyer, Nevin Liber, Nathan Meyers, Roger Orr, Geoff Romer, Patrice Roy, Tony van Eerd and Ville Voutilainen for suggestions and useful discussion.

References

[N3339] "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>

[S.Parent] "C++ Seasoning", Sean Parent, 2013

<https://github.com/sean-parent/sean-parent.github.io/wiki/Papers-and-Presentations>

[Impl] Reference implementation: `polymorphic_value`, J.B.Coe

<https://github.com/jbcoe/polymorphic_value>

[P0302r1] "Removing Allocator support in std::function", Jonathan Wakely

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0302r1.html>