A polymorphic value-type for `C++`

ISO/IEC JTC1 SC22 WG21 Programming Language `C++`

P0201R2

Working Group: Library Evolution

Date: 2017-10-16

Jonathan Coe <jonathanbcoe@gmail.com>

Sean Parent <sparent@adobe.com>

Change history

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:

```
// Copyable composite with mutable polymorphic components
class CompositeObject {
  std::polymorphic_value<IComponent1> c1_;
  std::polymorphic_value<IComponent2> c2_;

 public:
  CompositeObject(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`. The compiler-generated `compiler-generated destructor, copy, move, and assignment operators of`CompositeObject`behave correctly.`
`In addition to enabling compiler-generation of functions for composite objects,`polymorphic_value`performs deep copies of`c1_`and`c2_‘
without the class author needing to provide a special ‘clone’ method.

Deep copies

To allow correct copying of polymorphic objects, `polymorphic_value` uses the copy constructor of the derived-type pointee when copying a base type `polymorphic_value`. Similarly, to allow correct destruction of polymorphic component objects, `polymorphic_value` uses the destructor of the derived-type pointee 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 pointer
polymorphic_value<Base> poly(new Derived());
// AND the polymorphic_value to base is copied.
auto poly_copy = poly;

// THEN the copy points to a distinct object
assert(&*poly != &*poly_copy);
// AND the copy points to 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 desctructor 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.

Custom allocators

Custom allocators are not explicitly supported by `polymorphic_value`. Additional constructor(s) along with custom copiers and deleters can be added to support custom allocators. The specification the the additional constructors and copiers would depend on whether the allocator is to be used for only internal use or for allocation of the managed object too.

Design changes from `cloned_ptr`

The design of `polymorphic_value` is based upon `cloned_ptr` after advice from LEWG. The authors would like to make LEWG explicitly aware of 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 the cast operations provided for `cloned_ptr`.

[Should we be standardizing vocabulary types (`optional`, `variant` and `polymorphic_value`) or components through which vocabulary types can be trivially composed (`propagate_const`, `cloned_ptr`)?]

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__constru

```
namespace std {
class bad_polymorphic_value_construction : std::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;`

- 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]

A *polymorphic__value* is an object that owns another object and manages that other object through a pointer. More precisely, a `polymorphic value` is an object `v` that stores a pointer to a second object `p` and will dispose of `p` when `v` is itself destroyed (e.g., when leaving block scope (9.7)). In this context, `v` is said to own `p`.

A `polymorphic_value` object is empty if it does not own a pointer.

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 customised by supplying a copier and deleter.

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

The template parameter `T` of `polymorphic_value` may not be an array type.

The template parameter `T` of `polymorphic_value` may not be a function pointer.

[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; // see below

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

  polymorphic_value(const polymorphic_value& p);
  template <class U> polymorphic_value(const polymorphic_value<U>& p); // see below
  polymorphic_value(polymorphic_value&& p) noexcept;
  template <class U> polymorphic_value(polymorphic_value<U>&& p); // see below
```

```
  template <class U> polymorphic_value(U&& u); // see below

  // Destructor
  ~polymorphic_value();

  // Assignment
  polymorphic_value &operator=(const polymorphic_value& p);
  template <class U>
    polymorphic_value& operator=(const polymorphic_value<U>& p); // see below
  polymorphic_value &operator=(polymorphic_value &&p) noexcept;
  template <class U>
    polymorphic_value& operator=(polymorphic_value<U>&& p); // see below

  template <class U>
    polymorphic_value& operator=(U&& u); // see below


  // Modifiers
  void swap(polymorphic_value<T>& 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 ...Ts> polymorphic_value<T>
  make_polymorphic_value(Ts&& ...ts); // see below

// 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;
```

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

- *Postconditions*: `bool(*this) == false`

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

- *Effects*: 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 is moved from `c` and `d`.
- *Requires*: `C` and `D` are copy constructible, nothrow destructible and nothrow moveable. 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 `U` must be the same.
- *Throws*: `bad_polymorphic_value_construction` if `std::is_same<C, default_copy<U>>::value`, `std::is_same<D, default_delete<U>>::value` and `typeid(*u)!=typeid(U)`.
- *Postconditions*: `bool(*this) == bool(p)`.
- *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` initialised with this constructor.

```
polymorphic_value(const polymorphic_value &p);
template <class U> 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`. The copy is created by the copier in `p`.
  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`.
- *Postconditions*: `bool(*this) == bool(p)`.

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

- *Remarks*: The second constructor shall not participate in overload resolution unless `U*` is convertible to `T*`.
- *Effects*: Move-constructs a `polymorphic_value` instance from `p`. 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`.
- *Postconditions*: `*this` contains the old value of `p`. `p` is empty.

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

- *Remarks*: Let `V` be `std::remove_cv_t<std::remove_reference_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))`.

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

- *Remarks*: The second function shall not participate in overload resolution unless `U*` is convertible to `T*`.
- *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`.
- *Returns*: `*this`.
- *Postconditions*: `bool(*this) == bool(p)`.

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

- *Remarks*: Let `V` be `std::remove_cv_t<std::remove_reference_t<U>>`. This function shall not participate in overload resolution unless `V>` is not a specialization of `polymorphic_value` and `V*` is convertible to `T*`.
- *Effects*: the owned object of `*this` is initialised with `V(std::forward<U>(u))`.
- *Returns*: `*this`.
- *Postconditions*: `bool(*this) == bool(p)`.

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

- *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 `this`. If `p` has a custom copier and deleter then the copier and deleter of `*this` is the same as those in `p`.
- *Returns*: `*this`.
- *Postconditions*: `*this` contains the old value of `p`. `p` is empty.

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

```
void swap(polymorphic_value<T>& 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 ...Ts> polymorphic_value<T>
  make_polymorphic_value(Ts&& ...ts);
```

- *Returns*: A `polymorphic_value<T>` owning an object initialised with `T(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, David Krauss, Thomas Koeppe, Nevin Liber, Nathan Meyers, Roger Orr, Patrice Roy, Tony van Eerd and Ville Voutilainen for 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>`