An indirect value-type for C++

ISO/IEC JTC1 SC22 WG21 Programming Language C++
P0201R1
Working Group: Library Evolution
Date: 2016-10-13
Jonathan Coe <jbcoe@me.com>
Sean Parent <sparent@adobe.com>

Change history

Changes since P0201R1:
• Change name to indirect.
• Remove static_cast, dynamic_cast and const_cast as indirect 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.

Changes since P0201R0:
• Added a specialization of propagate_const.
• Added support for custom copiers and custom deleters.
• Removed hash and comparison operators.

TL;DR

Add a class template, \texttt{indirect<T>}, to the standard library to support free-store-allocated objects with value-like semantics.

Introduction

The class template, \texttt{indirect}, confers value-like semantics on a free-store allocated object. An \texttt{indirect<T>} may hold a an object of a class publicly derived from T, and copying the indirect 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:

```cpp
// Simple composite
class CompositeObject_1 {
  Component1 c1_;  // oops! (yes, it's Component1)
  Component2 c2_;  // oops! (yes, it's Component2)

public:
  CompositeObject_1(const Component1& c1, const Component2& c2) :
    c1_(c1), c2_(c2) {}  // oops! (yes, c was intended)

  void foo() { c1_.foo(); }  // no, it's not above
  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).

```cpp
// Non-copyable composite with polymorphic components (BAD)
class CompositeObject_2 {
  IComponent1* c1_;  // oops! (yes, it's IComponent1)
  IComponent2* c2_;  // oops! (yes, it's IComponent2)

public:
  CompositeObject_2(const IComponent1* c1, const IComponent2* c2) :
    c1_(c1), c2_(c2) {}  // oops! (yes, c was intended)

  void foo() { c1_->foo(); }  // no, it's not above
  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_) {  // oops! (yes, c was intended)
    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:

```cpp
// 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].

```cpp
// Copyable composite with immutable polymorphic components 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 indirect a copyable composite object with polymorphic components can be written as:

// Copyable composite with mutable polymorphic components
class CompositeObject_5 {
std::indirect<IComponent1> c1_
std::indirect<IComponent2> c2_

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

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

};

CompositeObject_5 has a (correct) compiler-generated destructor, copy, move, and assignment operators. In addition to enabling compiler-generation of functions, indirect 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, indirect uses the copy constructor of the derived-type pointee when copying a base type indirect. Similarly, to allow correct destruction of polymorphic component objects, indirect uses the destructor of the derived-type pointee in the destructor of a base type indirect.

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 : Base { void foo() const override {} }

// WHEN an indirect to base is formed from a derived pointer
indirect<Base> dptr(new Derived());
// AND the indirect to base is copied.
auto dptr_copy = dptr;

// THEN the copy points to a distinct object
assert(&*dptr != &*dptr_copy);
// AND the copy points to a derived type.
assert(dynamic_cast<Derived*>(&*dptr_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 \texttt{shared\_ptr} already implements
deep-destruction without needing virtual destructors. deep-destruction and
deep-copying can be implemented using type-erasure [Impl].

\section*{Lack of hashing and comparisons}

For a given user-defined type, \( T \), there are multiple strategies to make
\texttt{indirect<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 \texttt{indirect} 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 \texttt{std::hash<indirect<T>}}
and implement comparison operators for \texttt{indirect<T>}. 

\section*{Custom copiers and deleters}

The resource management performed by \texttt{indirect} - copying and destruction of
the managed object - can be customized by supplying a \texttt{copier} and \texttt{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 \texttt{unique\_ptr}.

We define the default copier in technical specifications below.
Custom allocators

Custom allocators are not explicitly supported by indirect. Since all memory allocation and deallocation performed by indirect is done by copying and deletion, any requirement for custom allocators can be handled by suitable choices for a custom copier and custom deleter.

Design changes from cloned_ptr

The design of indirect 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.

indirect<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. indirect can be built from cloned_ptr and propagate_const but there is no way to remove const propagation from indirect.

As indirect is a value, dynamic_pointer_cast, static_pointer_cast and const_pointer_cast are not provided. If an indirect 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 indirect) 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_copier [default.copier]

X.X.1 Class template default_copier general [default.copier.general]

The class template default_copier serves as the default copier for the class template indirect.

The template parameter T of default_copier may be an incomplete type.
X.X.2 Class template default_copier synopsis [default.copier.synopsis]

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

X.X.3 Class template default_copier [default.copier]

T* operator()(const T& t) const;
• Returns: new T(t).

X.Y Class template indirect [indirect]

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

An indirect is an object that owns another object and manages that other object through a pointer.

When an indirect is constructed from a pointer to an object of type U, a copier c, and a deleter d, then a move-constructed copy of the copier and deleter is stored in the indirect along with the pointer u.

When copies of an indirect are required due to copy construction or copy assignment, copies are made using c(*u).

An indirect object is empty if it does not own a pointer.

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

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

namespace std {
    template <class T> class indirect {
        public:
            typedef T element_type;

            // Constructors
            indirect() noexcept; // see below
            template <class U, class C=default_copier<U>, class D=default_deleter<U>>
                explicit indirect(U* p, C c=C{}, D d=D{}); // see below
            indirect(const indirect& p);
            template <class U> indirect(const indirect<U>& p); // see below
indirect(indirect&& p) noexcept;
template <class U> indirect(indirect<U>&& p); // see below

// Destructor
~indirect();

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

// Modifiers
void swap(indirect<T>& p) noexcept;

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

// indirect creation
template <class T, class ...Ts> indirect<T>
make_indirect(Ts&& ...ts); // see below

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

// indirect I/O
template<class E, class T, class Y>
basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os,
const indirect<Y>& p);

} // end namespace std

X.Y.3 Class template indirect constructors [indirect.ctor]

indirect() noexcept;

- **Effects**: Constructs an empty indirect.
- **Postconditions**: bool(*this) == true
template <class U, class C=default_copier<U>, class D=default_deleter<U>>
explicit indirect(U* p, C c=C{}, D d=D{});

- **Effects:** Creates a indirect object that owns the pointer p and has a
move-constructed copy of both c and d.

- **Preconditions:** c and d are copy constructible. The expression c(*p) shall
return an object of type U*. The expression d(p) shall be well formed,
shall have well defined behavior, and shall not throw exceptions. Either U
and T must be the same type, or the dynamic and static type of U must
be the same.

- **Postconditions:** operator->() == p

- **Throws:** bad_alloc, or an implementation-defined exception when a re-
source other than memory could not be obtained.

- **Exception safety:** If an exception is thrown, d(p) is called.

- **Requires:** U is copy-constructible.

- **Remarks:** This constructor shall not participate in overload resolution
unless U is derived from T.

indirect(const indirect &p);

- **Remarks:** The second constructor shall not participate in overload resolu-
tion unless U is derived from T.

- **Effects:** Creates a indirect object that owns a copy of the object managed
by p.

- **Postconditions:** bool(*this) == bool(p).

indirect(indirect &&p) noexcept;

- **Remarks:** The second constructor shall not participate in overload resolu-
tion unless U* is convertible to T*.

- **Effects:** Move-constructs a indirect instance from p.

- **Postconditions:** *this shall contain the old value of p. p shall be empty.
p.get() == nullptr.

X.Y.4 Class template indirect destructor [indirect.dtor]

~indirect();

- **Effects:** d(u) is called.
X.Y.5 Class template indirect assignment [indirect.assignment]

indirect &operator=(const indirect &p);
template <class U> indirect &operator=(const indirect<U>& p);
• Remarks: The second function shall not participate in overload resolution unless U* is convertible to T*.
• Effects: *this shall own a copy of the resource managed by p.
• Returns: *this.
• Postconditions: bool(*this) == bool(p).

indirect &operator=(indirect&& p) noexcept;
template <class U> indirect &operator=(indirect<U> &&p);
• Remarks: The second constructor shall not participate in overload resolution unless U is derived from T.
• Effects: Ownership of the resource managed by p shall be transferred to this.
• Returns: *this.
• Postconditions: *this shall contain the old value of p. p shall be empty.

X.Y.6 Class template indirect modifiers [indirect.modifiers]

void swap(indirect<T>& p) noexcept;
• Effects: Exchanges the contents of p and *this.

X.Y.7 Class template indirect observers [indirect.observers]

const T& operator*() const;
• Requires: bool(*this).
• Returns: A reference to the owned object.

const T* operator->() const;
• Requires: bool(*this).
• Returns: A pointer to the owned object.

T& operator*();
• Requires: bool(*this).
• Returns: A reference to the owned object.

T* operator->();
• Requires: bool(*this).
• Returns: A pointer to the owned object.

explicit operator bool() const noexcept;
• Returns: false if there is no owned object, otherwise true.

X.Y.8 Class template indirect creation [indirect.creation]
template <class T, class ...Ts> indirect<T>
    make_indirect(Ts&& ...ts);
    □ Returns: indirect<T>(new T(std::forward<Ts>(ts)...));

X.Y.9 Class template indirect specialized algorithms [indirect.spec]
template <typename T>
    void swap(indirect<T>& p, indirect<T>& u) noexcept;
    □ Effects: Equivalent to p.swap(u).

X.Y.10 Class template indirect I/O [indirect.io]
template<class E, class T, class Y>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os,
        const indirect<Y>& p);
    □ Effects: os << p.get().
    □ Returns: os.

Acknowledgements

The authors would like to thank Maciej Bogus, Germán Diago, Bengt Gustafsson,
David Krauss, Nevin Liber, Nathan Meyers, Roger Orr, Sean Parent, Patrice
Roy 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>
Reference implementation: indirect, J.B. Coe
<https://github.com/jbcoe/indirect>