Smart References through Delegation (2nd revision)

Project:	ISO/IEC JTC 1/SC 22/WG 21/C++
Document #:	P0352R1
Date :	2017-02-06
Revises :	P0352R0
Working Group:	Evolution

Reply to:	Hubert Tong hubert.reinterpretcast@gmail.com

Hubert Tong
Faisal Vali

Abstract

We propose an inheritance-like mechanism, suitable for the implementation of a smart reference. The smart reference type would take the form of a class that "inherits" from the referred-to type. As with classic inheritance, a "derived class" instance would be useable as an instance of the "base class". The idea is that, under this mechanism, what corresponds to the base class subobject is not automatically allocated or initialized as part of a most derived object; but instead, a conversion function determines how an object of the "base class" type is obtained. Member lookup (in all its multiple inheritance glory) simply works as it currently does. Thus, we get all the benefits and familiarity of member lookup through the established rules for (multiple-)inheritance without its dreaded disadvantage of linking the object layouts.

Summary of Changes

From P0352R0:

- This is a cosmetic update of the paper that does not introduce any new syntax or functionality (as guided by the discussion in EWG)
- Updated select references to N4477 to refer instead to P0416R1
- Updated discussion to better reflect the state of Operator Dot in P0416R1/P0252R2
- In response to feedback from Issaquah, recast the new feature, not as a form of inheritance, but as as inheritance-like mechanism
- In response to feedback from Issaquah, changed the name of the feature to avoid "delegate" (since it is overloaded in some dialects of C++)
    - We are open to other suggestions for the naming
- Added a brief update to the Core Wording section

Table of Contents

1 Motivation

The background and motivation for a core language feature that supports implementing smart references and proxies in C++, is described by our esteemed friends in their excellent paper P0416R1: Operator Dot (R3)[1]. We refer readers in need of further background to that paper, since we agree with the value placed on idioms that limit raw pointer use (without compromising efficiency) and recognize the importance of the proxy pattern[2]. Generally, we agree that the problems that P0416 aims to solve need to be solved.

But, given that P0416 solves the problem it aims to solve, readers have to wonder why the current authors engineered an alternative proposal to solve those same problems.

In brief, we were motivated by feedback about N4477 (a previous incarnation of Operator Dot).

For those who are unfamiliar with the operator dot strategy for supporting smart-references and proxies; please read P0416 and its wording paper, P0252, we can not do it justice in this space. For those who just want a brief and incomplete sketch as a refresher, we formulated the following:

> P0416/P0252 proposes overloading operator dot to implement smart-references, so that any implicit or explicit use of operator dot (with some exceptions, such as through use of the arrow operator on a pointer) on an object of a type that overloads operator dot - incurs member lookup into the return types of all overloaded operator dot member functions, should a name (including inherited ones) not be found in the parent type itself. Once such a member name is found, the "operator dot" is used to form an implicit-conversion-sequence to ***convert*** the object-expression of the member-access to the desired containing type. This extends to the return types of the overloaded dot operators themselves, such that the conversion may involve a sequence of operator-dots.

Discussion within the Core Working Group has raised questions with the above design (the feature seems more coherent with a "conversion function" as opposed to a member-access operator-dot—member lookup across dot-operators attempt to model multiple inheritance—and regarding the complexity of overloading on the cv-qualification & value-category of the object-expression) during its initial review of P0252R0 (wording for N4477) in Jacksonville[3] and the

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0416r1.pdf
[2] https://sourcemaking.com/design_patterns/proxy/cpp/1, https://en.wikipedia.org/wiki/Proxy_pattern
[3] http://wiki.edg.com/bin/view/Wg21jacksonville/CoreWorkingGroup#P0252R0_Operator_Dot_Wording
Richard: Did EWG discuss expressing this as a conversion operator, instead of operator.?
… Richard: We should strive to keep operator-> and operator. the same.
… Gaby: Tries to model multiple inheritance.
… John: Where do you look up the first part of the qualified-id? p->A::x; where do you look for A?
Gaby:

```
template <class T>
```

BSI C++ Panel has raised concerns[4] about its complexity, since. Additionally, Botond Ballo, in his trip report following Kona, reported on other potential objections having to do with reflection that were discussed at the meeting[5].

Nevertheless, the authors of this paper feel strongly - similar to the authors of P0416 - that the smart-reference and proxy problems are in need of urgent solving.

While we recognize that the authors of P0416 might still be able to address any concerns raised with their design (to the satisfaction of the committee) – since we can not predict any given plenary's outcome, we simply admit to being sensitive to the already-raised concern by a National Body; and therefore, in an effort to maximize the chance of securing a feature that supports expressing smart-references and proxies elegantly in C++, and hoping to build on all the hard work of the authors of P0416 (in analyzing the solution space, delineating an operator-dot design and generating valuable feedback) – we present an alternative proposal to those who are concerned by the operator-dot strategy for smart-references.

At the outset, we should admit that our design philosophy involved defining a set of coherent sub-features with non-monolithic responsibilities that were as clear and as simple as possible – while being composable and idiomatic – with the hope that through their proper composition one could express the examples and use-cases from P0416 with less implied or imagined complexity.

```
struct ref {
  T& operator.();
  void reset(T&);
};
ref<A> r;  r.reset(a);
r.B::a;   // lookup B inside ref (nothing found), then lookup B in decltype(r.operator.())
```

John: The fact that a given B may not have an x member, doesn't matter?

http://wiki.edg.com/bin/view/Wg21jacksonville/CoreWorkingGroup#P0252R0_Operator_Dot_Wording_AN1

… Richard: This is not an operator. , it's a very implicit conversion operator.

… Richard: This has the semantics of an implicit conversion operator; this isn't just applied to "dot" in source code.

[4] http://lists.isocpp.org/core/2016/04/0321.php "We are not persuaded that the use cases in this proposal (and its predecessor papers) outweigh the additional complexity in the language. At the end of our discussion we took a straw poll: Will we want to raise a NB objection if this proposal is moved to the C++17 WP? SF 5/WF 3/N 3/WA 1/SA 0.

[5] https://botondballo.wordpress.com/2015/11/09/trip-report-c-standards-meeting-in-kona-october-2015/

In preparing for an alternative design, we started with an (admittedly incomplete, given our time constraints) analytic attempt at identifying the potential sources of complexity (real or imagined) in N4477/P0416 – so that we could deconstruct and strategize around them – which led us to the following list [*updated for P0416R1/P0252R2*]:

1. When invocation of an overloaded operator dot does or does not occur might surprise some programmers:

    - expression `++a` does not lexically contain operator 'dot', yet would invoke operator-dot (`becomes: a.operator.().operator++())`

    - `(&a)->operator++()` would not invoke operator-dot (`becomes: a.operator++())`

    - `(*&a).operator++()` would invoke operator-dot (`a.operator.().operator++())`

    - The need to potentially resort to 'addressof()' when referring to non-static-members within member-functions, to avoid recursive calls to operator-dot (also since use of *id-expression*s can get transformed to member-accesses (*this).*id-expression* one might need to be careful):
      ```
      Ref(Ref&& a) : p{std::addressof(a)->p} { std::addressof(a)->p = 0; }
      ```

        - Note: operator-dot overloads can bypass members of the smart-reference type in P0416; P0252 gives special mention to members declared `public`.

2. The lack of equality between `p->foo()` and `(*p).foo()` for a raw pointer 'p'; if `(*p)` is of a type that overloads operator-dot – could bother some seasoned programmers.

3. That overload resolution does not consider candidates from multiple sources might be deemed unnecessarily limiting:

    - Some users might prefer the option of hiding while others might prefer having an overload across all declarations through member *using-declaration*s.

4. Operator-dot being called implicitly when forming implicit-conversion-sequences (such as during overload resolution), like a conversion operator, might take some getting used to.

5. An overloaded operator-dot can return a builtin-type that one can NOT use the dot operator on, but with unified function call, auto-type deduction and the fact that operator-dot is invoked to convert the object argument during overload-resolution, might serve some useful purpose:

    ```
    struct A { int operator.(); }; // OK?
    ```

    - This is inconsistent with the 'operator->' and might surprise some folks

6. Cannot get access to member type names easily through operator-dot. [P0416R1 §4.15 Member Types]

7. Defining the semantics of member lookup through a sequence of operator dots would complicate member lookup further, along with specifying how cv-qualification and value-category of the left hand side (i.e., the object expression) should affect the overload-resolution of a sequence of operator-dot invocations. This attempts to model a form of multiple inheritance.

With consideration for the fact that inheritance[6] is a well-established[7] method of exposing and extending class interfaces, and that conversion functions are well-recognized for their use in forming implicit-conversion-sequences (as opposed to operator-dot), we felt that a model based on inheritance and conversion operators would be more familiar to C++ programmers. In addition, the member name lookup rules would require no changes, leveraging the already existing complexity for ambiguity resolution in the presence of multiple inheritance.

Briefly, the idea is that class may be declared to *adapt* a class in a manner similar to declaring inheritance from a base class. Such an *adapted class* acts much like a base class; however, it does not affect the layout of the adapting class. That is, it does not introduce a "base class subobject" to be automatically allocated or initialized as part of a most derived object; but instead, an *adapting conversion function* (as opposed to an overloaded operator-dot) determines how to provide an instance of the adapted class for operations, and member lookup (in all its multiple inheritance complexity) simply works as it currently does.

[6] A model based on inheritance was considered in D&E 12.7, raising concerns, which we feel we address:

- Since we allow the adapting class to hide functions from the class being adapted, we address the chief "cause of bugs and confusion" in that model
- Unless the adapted class is specifically designed (e.g., by passing in a pointer) to rely on the adapting class (i.e. "derived/inheriting" class), it shouldn't care about using functions from the adapting class.

We also note that Golthwaite's N1363's delegation approach is really more about renaming/forwarding as opposed to our approach.

[7] It is noted that there is guidance against the use of inheritance, e.g., objections over the "strong coupling" implied by a inheritance-relationship. This proposal attempts to address that concern by introducing a weaker form of inheritance. The authors further believe that the presence of guidance over the use of inheritance is a point in favour of this proposal: there is immediately a larger knowledge base for members of the community to draw from.

While we go into details in the later sections of the paper, here's an early glimpse into the idea's look and feel, as compared to P0416:

Current Proposal

```
template<class X> class Ref : public using X {
  X* p;
public:
   operator X&() { /* maybe some code here */ return *p;
}
  explicit Ref(int a)      : p{new X{a}} {}
  ~Ref()                   { delete p; }
  void rebind(X* pp)       {
    if (p != pp) { delete p; p = pp; }
  }

  // …
};

struct Y { Y(int); void f(); };

Ref<Y> r {99};

r.f();          // OK: means (r.operator Y&()).f()

Ref<Y> &r2 = r; // OK

void  g(Y  &y);    //  Namespace  scope  name  is  always
'public'
g(r);          // OK: g(r.operator X&())

Y y = r;       // OK
```

P0416 Equivalent

```
template<class X> class Ref {
  X* p;
public:
  X& operator.() { /* maybe some code here */ return *p; }
  explicit Ref(int a)    : p{new X{a}} {}
  ~Ref()                 { delete p; }
  void rebind(X* pp) {
    if (p != pp) { delete p; p = pp; }
  }

  // …
};

struct Y { Y(int); void f(); };

Ref<Y> r {99};

r.f();          // OK: means (r.operator.()).f()

Ref<Y> &r2 = r; // OK

void g(Y &y);
g(r);           // OK: g(r.operator.())

Y y = r;        // OK:
```

2 Précis

In terms isomorphic with N4477/P0416, our proposal can be thought of broadly in three composable parts:

1. **The Adapted Class**: We propose taking the return type of the operator-dot, and instead naming it within what is now the base specifier list. The advantage of this strategy is that it does not require any change to the already complicated member lookup rules, when looking up names in classes - that is, we do not attempt to 'model multiple inheritance' as N4477's wording and design seems to do [See Gaby's comment in Core] - we simply leverage the already existing model for member-lookup when multiple bases are specified.

```
struct A; struct B;  // Note: adapted classes may be incomplete.

struct C : using A, using B { }; // Adapted classes do not affect layout of C.

C cobj; // Can create a C object without requiring A or B to be complete.

// If name lookup through C must look 'into' its adapted classes, those classes
// would need to be complete. Semantics of constructs shall be the
// same regardless of where and when types are completed; no diagnostic
// required.
struct A { static void f(); };
struct B { static void g(); using T = int*; };

cobj.f(); // OK, A::f(), no conversion to implicit object parameter for static
C::T ip = 0; // OK [does NOT work with N4477]
```

2. **The Adapting Conversion Operator**: We propose using a conversion function – instead of an 'operator-dot' – to express the conversion from the object-expression to an adapted class. This conversion operator would be invoked when converting the object-expression to the implicit object parameter of the eventual member function being invoked, or to the type that declared the non-static data-member. The advantage is that a **conversion function** is used to do a conversion as opposed to 'operator-dot' being used to do a conversion. The conversion exists regardless of the accessibility of the conversion function. A possible interpretation of the accessibility of the conversion function is that it applies to conversions not bound to a member access; that is, when a member is being accessed, the accessibility of the conversion function is ignored.

```
struct A { void f(); int data; } sharedA {};
struct B : using A {
  operator A&() { return sharedA; };
} b;

// Name Lookup and overload resolution select non-static A::f. Its
// implicit object parameter is of type A&.  The conversion from
// the object argument 'b' of type 'B' to 'A&' invokes conversion
// operator 'operator A&' to adapted class.

b.f();  // OK: after lookup finds 'A::f' and overload resolution selects it,
        // becomes (b.operator A&()).f();


b.data = 5; // OK: becomes (b.operator A&()).data
```

Thus, the equivalent implementation of a smart reference as described in P0416, using this proposal, would look like:
(Notice the similarity to an attempt by a User on Stack Overflow in 2009[8])

```
template<class X> class Ref : public using X {
public:
  Ref(X &r) : p{&r} {}
  ~Ref() { delete p; }
  void rebind(X &r) {
    if (p != &r) { delete p; p = &r; }
  }
  Ref<X> &operator=(X &r) { *p = r; return *this; } // mimic built-in reference
  // …
private:
  X *p;
  // Here, the conversion function inherits access of the looked-up name
  // (and namespace functions under uniform function call syntax have
  // 'public' access by default)
  // when invoked implicitly for converting the object-expression to become
  // the implicit object-argument.  But when called directly, e.g., to bind
  // to a reference declaration it gets its access from here (i.e. private).
  //
  // Access to the inherited members can be achieved using an
  // access-specifier in the corresponding base-specifier.
  operator X&() { /* maybe some code here */ return *p; }
};
struct Y { Y(int); void f(); };

// All behavior below should be similar to N4477's.
Ref<Y> r{99};
r.f();          // OK - invokes y.f(); conversion inherits access of 'f'
r = Y{9};       // OK - invokes Ref<Y>::operator=(Y&)
Ref<Y> &r2 = r; // OK
auto r3 = r;    // OK - decltype(r3) == Y
Y &y = r;       // Error - conversion function is private
void g(Y &y);   // Namespace scope operation.
r.g(); // uniform call syntax: conversion has access of 'g' (i.e. public)
g(r); // Required by N4477 - conversion has access of 'g' (i.e. public)
```

[8] http://stackoverflow.com/questions/1307876/how-do-conversion-operators-work-in-c/

3 Details and Technicalities

We simply list some of the details and technicalities[9], which we would be happy to expand upon further, should the committee indicate an eventual desire:

1. Adapting a class does not affect the adapting class's layout or size or composition, unlike inheriting from a base class; therefore, there is no fundamental reason for it to factor into the standard-layout-ness, trivial-ness, POD-ness, aggregate-ness, literal-ness etc. of the derived type. Nevertheless, we cautiously recommend that the presence of an adapted class restricts these properties for now, and perhaps consider lifting the restrictions as the need arises.

2. While there is no fundamental reason Unions can't adapt classes, or be adapted, one again we cautiously recommend restricting these for now, and consider lifting the restrictions as the need arises.

3. The only fundamental restrictions on use of an adapted class as opposed to a base class are:
    a. Unlike a layout base class, the conversion performed by a static_cast from the adapted ("base") to the adapting ("derived") type is ill-formed; note: the C-style cast is similarly ill-formed and does not fall back to reinterpret_cast
    b. it can not be designated virtual ('virtual' collapses the subobjects associated with the bases into one, and since adapted classes don't by themselves have a unique object in relation to the adapting class, but rather allow the user to programmatically provide an appropriate object in lieu, 'virtual' is unnecessary). Yet, there might be some useful semantics for "virtual" adaptation that we mention in our section on Future directions.
    c. can not "inherit" constructors from an adapted class or otherwise initialize the adapted class as a base class (e.g., by a mem-initializer)—there is also no similar compiler-generated initialization
    d. virtual functions of an adapted class are not considered when trying to determine the functions being overriden in an adapting class

```
struct B { virtual void v(); };
struct C : B { void v() override; }; // OK
struct D : using C { void v() override; }; // error: v() does not override anything
```

[9] While we've thought through some of these issues - and can discuss this further if the paper makes it to CWG, for now we don't have the time to do the details justice through proper exposition.

4. A built-in type can be an adapted class – this is useful for allowing smart-references to contain built-in types without having to resort to partial specializations - such a class would behave very similar to any other class today that contains an implicit conversion operator to the built-in. For the most part, the conversion operator itself would endow the smart-reference with all the desired properties - the only functionality that being an adapted class adds to the conversion-operator is for uniform-function-call syntax - where name lookup for uniform call syntax would consider the conversion.

5. Adapted classes can be public, private or protected, just like layout bases.

6. A type can adapt a class that itself adapts another class, mingled with layout bases within a class lattice. For each adaptation relationship, a conversion to the adapted class represents a potential call to an adapting conversion function.

7. An adapted class does not need to be complete in a TU, unless name lookup within the class or after its definition, requires delving into the adapted classes (i.e., name is not found in the adapting class); the program is ill-formed if the semantics differ depending on whether the class is complete or not (e.g., if lookup would find the name through the adapted class in one case and find the name in a namespace scope in the other).

8. OK for an adapted class to be a "final" class

9. Using-member declarations designating members from adapted classes work

10. Static members of an adapted class should be callable/usable without requiring or going through a conversion operator (no empty base optimization necessary)

11. Implicit conversion from a pointer-to-adapting 'dp', to a pointer-to-adapted '*cv* B' is performed as ({ *cv* B &__b = *dp; &__b; }).

12. It is probably bad if an inherited conversion operator acts as an adapting conversion operator.

13. A pointer to member of an adapted class cannot be converted to a pointer to member of the adapting class.

```
struct A { int x; };  struct B : using A { using A::x; };
int A::*px = &B::x; // OK; note: &B::x gives a pointer-to-member of A
int B::*px2 = &B::x; // NOT ok
```

4 Implementation

An implementation using Clang is being worked on.

5 Teachability

We expect this to be relatively easy to teach to programmers who are familiar enough with inheritance, composition and user-defined conversions - since adaptation of classes can be thought of as a hybrid of the two approaches (inheritance and composition) for reuse. For novices and unseasoned C++ programmers, an integrated approach in introducing inheritance and adaptation may be taken: with adaptation, the "derived" class no longer owns a "base-class" instance, but instead borrows its interface and implementation through an adapting conversion function; this borrowing becomes the common ground between inheritance and adaptation. The authors look forward to the insights that may come from the new crop of C++ programmers who cut their teeth in this new world.

6 Core Wording

We shall await feedback from EWG before presenting core wording that would be appropriate for a CWG audience. In brief, adaptation would be added alongside inheritance. Where inheritance relies on being able to perform conversions based on known layout relationships, adaptation would allow conversions only when an adapting object is available for a call to an appropriate adapting conversion function. For the purposes of polymorphic behavior, an adapted class has no special relationship to an adapting class. Primarily, the wording changes will be for adaptation to opt-in to inheritance-like behavior. In only limited cases, such as the reinterpret_cast fallback of a C-style casts, does extra wording come into play to avoid unwanted behavior.

7 Future Directions

1. Explicit adapting conversion operators:
   We could have explicit mean that the conversion is valid only in direct initialization (the function call is not such a case) and when the member is named using an appropriate qualified-name

2. Virtual adapted classes:
   Presently we give no semantics to virtual adapted classes, but these could be considered a promise from the user that conversion to the virtual adapted class may be achieved by arbitrarily choosing a path in the class lattice.

3. Lifting the non-fundamental restrictions mentioned in the section on Details and Technicalities.

4. Allowing deleted conversion operators to disown bases or adapted classes from unwanted name lookup and conversion.

5. It may be useful for there to be a way to deduce the type of the implied object argument.

6. It may be useful for there to be a way to duplicate the cv-qualification of a deduced type.

8 Pros and Cons

Advantages of Adaptation over P0416R1:

- ➔ using Adapted::f allows both Option 1 and Option 4 from N4477 [Section 4.4] to either hide a base name or bring its overloads into Derived scope as usual. One can not do this with operator dot to un-hide.
- ➔ Does not complicate member lookup further
- ➔ Access control changes do not modify name lookup (refer to [over.ref] changes in P0252R2)
- ➔ Does not require empty-base optimization when adaptation is used to access nested types, enumerators, etc. - since it does not add to/affect the class's size.
- ➔ Does not interfere at all with a potential future operator-dot proposal that supports intercession [P0060r0]
- ➔ Does not break the equality of the following expressions: `p->f() <==> (*p).f()` which N4477 purports to do if the type of (*p) overloads operator-dot - that is in N4477 if the member function call is written as p->f() it equates to (*p).f() (i.e. does not invoke operator-dot) BUT if it is written as (*p).f() it equates to (*p).operator.().f().
- ➔ Does not tax users with having to worry about the implicit invocation of operator-dot in expressions where we do not clearly see it lexically.
- ➔ Allows access to nested types through familiar inheritance behavior.

Advantages of N4477 Operator Dot (R2) over Adaptation:

- ➔ Member lookup and Overload resolution occurs twice for Adaptation: first when looking up the member name and second when finding the right member conversion operator from the adapting object type to the adapted target type (based on the value category of the object expression).
- ➔ Operator-dot can return nested classes defined within the body of the class that overloads operator-dot or local classes defined within operator-dot's body (with auto return type) - adaptation (since it is declared similarly to a base class) obviously require the types that are being adapted to be defined outside the class doing the adapting.
- ➔ Operator-dot provides a method to restrict lookup to names belonging to the Handle: using a pointer; this mostly only affects names which are not part of the Handle (to produce errors).

Neutral:

- ➔ Adaptation does not require additional friendship between the "handle" and the "implementation" for access to 'protected' members. The usual tricks for preventing inheritance from a class are also not entirely applicable to adaptation; therefore, the

dynamics of 'protected' access (or more generally, the handshake between base classes and derived classes) change.

➔ Both allow reference leaking through direct reference binding to function parameters - and both employ mechanisms to prevent direct reference binding to reference variables.

9 Acknowledgment/References

N4477, "Operator Dot (R2)", its revisions, and companion wording papers

The authors would like to thank Richard Smith, Hal Finkel, Jens Maurer, John Spicer—and any others who may have been unintentionally missed—for their feedback and encouragement. Any mistakes or misjudgments in this proposal are the responsibility of the authors.

Appendix A: Member Access through an Adapted Class In Depth

```
struct T {
  void t() &;
};

template<class T> class Wrap : public using T {
  T *p;
  operator T&() { return *p; }
  void w();
public:
  Wrap(T *p) : p(p) { }
};

template<class T> class Ref : using public Wrap<T> {
  T *p;
  operator Wrap<T>() { return {p}; }
public:
  void r();
  Ref(T *p) : p(p) { }
};

Ref<T> robj{new T{}};
robj.r();  // No conversion necessary
robj.w();  // Error: 'w' has private access, name is inaccessible.
robj.t();  // OK : all adapting conversion operators inherit 'public' access of name 't'

Analysis:
    -   robj.r(): member lookup finds 'r', implicit object parameter is 'R&' which is same type and
        value category as the left hand side of the object expression – access is public – done.

    -   robj.w() : member lookup finds 'w', overload resolution selects it (the viability is
        established as it is in the case of a normal inheritance relationship; the reference binding
        would succeed in that case as a derived-to-base Conversion) – but name is inaccessible - so we
        don't even check the access of the conversion function – ill-formed.

    -   robj.t() :
                - first member lookup for t occurs; resulting in a declaration-set {T::t(T&)}
                - then perform overload resolution:
                        - the viability is established as explained above for ‘w’
                - best viable function is 't(T&)', and it is accessible; proceed with the next step.

                - since overload resolution succeeds, attempt to convert the left hand side of the
                member access to the implicit object argument using one or more adapting conversion ops

                - use the implicit object parameter of t(T&) 'i.e. T&' as the target and the lhs of the
                member access (robj) as the source for a sequence of base conversion operators:

        - the sequence of adapted types to convert to is determined by the inheritance+adaptation
        graph:
                                                [ T ]
                                                  ^
                                                  |
                                                using
                                                  |
                                                [ Wrap<T> ]
                                                  ^
                                                  |
                                                using
                                                  |
                                                [ Ref<T> ]
```

```
  - if there is no unique path from the class type of the lhs of the member access to the type
needed for the implicit object argument, then the program is ill-formed; otherwise, (for 'robj'
we do have a unique path: Ref<T> -> Wrap<T> -> T, so proceed) [This is a similar problem to the
ambiguous base problem that all multiple-inheriting-programmers have grown to internalize]

  - the target of conversion is initially the implicit object parameter (i.e. T& t)

  - a base conversion operator is chosen from the type which most directly adapts the type
needed (call it D [in our specific example 'Wrap<T>' directly inherits from 'T']); overload
resolution for the adapting conversion operator is performed using an invented source with type
D, and the cv-qualification and value category of the lhs
        - i.e. Wrap<T> &invented; T& t = invented; (invokes 'invented.operator T&(Wrap<T> &)')

  - if D is not the class type of the lhs, then repeat with the implicit object parameter of
the selected base conversion operator as the new target of conversion:
        - Wrap<T> is not the type of the lhs (i.e. Ref<T>) so check against the implicit object
        parameter of the conversion 'operator T&(Wrap<T> &)', i.e. Ref<T> &rinvented; Wrap<T>
        &wt = rinvented;  and we get rinvented.operator Wrap<T>() which binds a prvalue to an
        lvalue reference per the usual rules for implicit object parameters
```