Smart References through Delegation: An Alternative to N4477's Operator Dot

Project:	ISO/IEC JTC 1/SC 22/WG 21/C++
Document #:	P0352R0
Date :	2016-05-30
Revises :	None
Working Group:	Evolution
Reply to:	Hubert Tong hubert.reinterpretcast@gmail.com

Hubert Tong
Faisal Vali

Abstract

Motivated by the recent feedback regarding 'N4477: Operator Dot (R2)'[1] from the CWG[2] and UK's BSI[3], this proposal suggests an alternative strategy for solving the smart-reference and proxy problems. Rather than overloading operator dot[4], we propose the creation of a weaker form of inheritance, suitable for the implementation of a smart reference: as a class that inherits from the referred-to type. The idea is that inheritance may be done "by delegation" – meaning that a base class object is not automatically allocated or initialized as part of a most derived object; but instead, a delegate-base conversion function (as opposed to an operator-dot) determines how that delegate-base class is addressed, and member lookup (in all its multiple inheritance glory) simply works as it currently does.

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4477.pdf
[2] https://isocpp.org/std/the-committee
[3] https://www.cxxpanel.org.uk/
[4] We are certainly not opposed to overloading operator dot in principle and recognize that depending on how operator dot is overloaded, it may allow for powerful intercession and reflection-based use-cases (P0060, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0060r0.html), that would NOT be easily solvable either though N4477 or through our proposed strategy of delegate-inheritance for implementing smart-references.

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 N4477: Operator Dot (R2)[5]. 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[6]. Generally, we agree that the problems that N4477 aims to solve need to be solved.

But, given that N4477 solves the problem it aims to solve, readers have to wonder why the current authors chose to devote any energy whatsoever to an eleventh-hour alternative proposal, to solve those same problems.

In brief, we are motivated by recently unearthed feedback about N4477 (now that it is understandably garnering wider scrutiny from constituents outside the Evolution Working Group).

For those who are unfamiliar with the operator dot strategy for supporting smart-references and proxies, please read N4477, 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:

> N4477 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 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. Additionally when using 'auto' to declare a variable of a type that

[5] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4477.pdf

[6] https://sourcemaking.com/design_patterns/proxy/cpp/1, https://en.wikipedia.org/wiki/Proxy_pattern

overloads operator-dot, 'auto' is replaced by the return type of the operator-dot.

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[7] and the BSI C++ Panel has raised concerns[8] 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[9].

Nevertheless, the authors of this paper feel strongly - similar to the authors of N4477 - that the smart-reference and proxy problems are in need of urgent solving.

While we recognize that the authors of N4477 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

[7] 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>
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.
[8] 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.
[9] https://botondballo.wordpress.com/2015/11/09/trip-report-c-standards-meeting-in-kona-october-2015/

authors of N4477 (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 N4477 with less implied or imagined complexity.

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 – so that we could deconstruct and strategize around them – which led us to the following list (References in square-brackets are to sections in N4477):

1. Implicit invocation of operator dot might surprise some programmers [1, 4.1, 4.5]:

    - 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-expressions* can get transformed to member-accesses (*this).*id-expression* one might need to be careful):
      `Ref(Ref&& a) : p{addressof(a)->p} { addressof(a)->p = 0; }`

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 [4.1, 4.5].

3. Leveraging operator-dot to guide auto-type deduction could be seen as overloading operator-dot with too much implicit functionality (complicating its interface) [4.2]:

    - `auto r = Ref<X>{1}` per N4477 deduces '`r`' to be of type '`X`' not '`Ref<X>`' since

`Ref<X>` overloads an operator-dot that returns `'X'`.

- it's not obvious what should happen if one has multiple overloaded operator dots, but that could disable or complicate auto-deduction for that type.

4. A name that is declared in the smart-reference (Handle) class *always* hides the name of any of its Referent's (Value) classes might be deemed unnecessarily limiting [4.3, 4.4]:

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

5. 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 [4.2, 4.7, 5]

6. 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. [4.7]:

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

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

7. If one defines operator-dot one must define or delete all the special operations [4.5.1]

8. Cannot get access to type names easily through operator-dot [4.10]

9. Defining the semantics of member lookup through a sequence of operator dots would complicate member lookup further (by allowing names to be pulled in from different declaration-scopes without need for a disambiguating member using-declaration and performing overload resolution across all of them is a new addition), along with specifying how the 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 [4.9].

10. Allowing `sizeof` on an object to return the size of the operator-dot's return type and not the size of the type of the object, could prove a surprising consequence of overloading operator-dot [6.1, see comment in last example]

With consideration for the fact that inheritance[10] is a well-established[11] 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 inheritance may be done "by delegation" – meaning that a base class object is not automatically allocated or initialized as part of a most derived object; but instead, a delegate-base conversion function (as opposed to an overloaded operator-dot) determines how that delegate-base class is addressed, and member lookup (in all its multiple inheritance complexity) simply works as it currently does.

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

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

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

[11] 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 N4477:

Current Proposal

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

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

Ref<Y> r {99};

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

r = Y{9};       // OK: means r.operator=(Y{99})

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

Y &yr = r;      // Error: conversion function is private

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

Y y = r;       // OK
auto a = Ref<Y>{5}; // OK: decltype(a) == 'Y'

// Because of 'using sizeof'.
static_assert(sizeof(Ref<Y>{5}) == sizeof(Y));
```

N4477 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()

r = Y{9};       // OK: means
(r.operator.()).operator=(Y{9})

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

Y &yr = r;      // Error: operator-dot is inaccessible??

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

Y y = r;        // OK:
auto a = Ref<Y>{5}; // OK: decltype(a) == 'Y'

// Because of operator-dot.
static_assert(sizeof(Ref<Y>{5}) == sizeof(X));
```

As a basis for further discussion and convenience we draw attention to certain properties necessary for a well-behaved smart reference (consistent with N4477), that we urge the reader to keep in mind:

Assume: `Ref<X> rx{5}; X x{5}; template<class T> void f(T &Ref);`

All functions that work with the referent '`X&`' should behave similarly with the smart reference '`Ref<X>`', Consider:

We should be able to write a smart reference so that f(x) is equal to f(rx)

```
X x1 = rx;      // should be okay since x1 = x is okay
auto ax = rx;   // 'ax' should be of type 'X'
auto &ar = rx;  // 'ar' should be of type 'X&' but this should be ill-formed
```

2 Précis

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

1. **The Delegate Base**: We propose taking the return type of the operator-dot, and instead naming it as a delegate base within 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: incomplete delegate bases are okay.

struct C : using A, using B { }; // Delegate bases 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 delegate bases, those bases
// 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 Conversion Operator to Delegate Base**: We propose using a conversion function – instead of an 'operator-dot' – to express the conversion from the object-expression to a delegate base. 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 base 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. It would inherit its access from the access of the name that was found. If declared private, it would not participate in direct reference binding of variables (but would for function parameters, consistent with expected behavior from N4477)

```
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 delegate base.

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
```

3. **The Auto Alias:** We propose uncoupling 'auto' deduction guides into a separate, simple orthogonal feature. We recommend supporting 'using auto = type' [similar to the approach in N4035[12] and as previously implemented[13]] to allow 'correct' deduction that 'sees' through ephemerals and proxies - when declaring auto variables (as opposed to automatically using operator-dot's return type). The advantage is that this would allow a user to clearly choose and specify a type (and avoid incurring ambiguity such as could occur if multiple operator-dots are overloaded) for auto-deduction, without tying it to operator-dot.

```
struct A { };
struct Ref : using A {
  using auto = A;
};
auto r = Ref{}; // decltype(r) == 'A'
```

Thus, the equivalent implementation of a smart reference as described in N4477, using this proposal, would look like:
(Notice the similarity to an attempt by a User on Stack Overflow in 2009[14])

[12] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4035.pdf
[13] https://github.com/faisalv/clang/commits/using-auto-conv
[14] http://stackoverflow.com/questions/1307876/how-do-conversion-operators-work-in-c/

```
template<class X> class Ref : public using X {
public:
  explicit Ref(X *p) : p{p} {}
  // use variadics & forwarding for general case below.
  explicit Ref(int a) : p{new X{a}} {}
  ~Ref() { delete p; }
  void rebind(X* pp) {
    if (p != pp) { delete p; p = pp; }
  }
  using auto = X;
  Ref<X>& operator=(X& r) { *p = r; return *this; }
  // …
private:
  X* p;
  // The conversion function inherits access of the looked-up name
  // (and namespace functions have 'public' access by default)
  // when invoked implicitly for converting the object-expression to become
  // the implicit object-argument.  But when called directly 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)
```

3 Details and Technicalities

Given the time constraints and the looming deadline for the mailing (i.e. tomorrow), we simply list some of the details and technicalities[15], which we would be happy to expand upon further, should the committee indicate an eventual desire:

1. Inheriting a base by delegation by itself does not affect the inheriting class's layout or size or composition; 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 a delegate base 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 have delegate-bases, or be delegate bases, once again we cautiously recommend restricting these for now, and consider lifting the restrictions as the need arises.

3. The only fundamental restrictions on a type being used as a delegate base are:
    a. the conversion performed by a static_cast from the delegate base to the derived type is ill-formed; note: the C-style cast is similarly ill-formed
    b. it can not be designated a virtual base ('virtual' collapses the subobjects associated with the bases into one, and since delegate bases don't by themselves have any associated object, but rather allow the user to designate a subobject thus 'virtual' is unnecessary). Yet, there might be some useful semantics for virtual delegate-bases that we mention in our section on Future directions.
    c. can not inherit constructors from a delegate base or otherwise initialize the delegate base as a base class (e.g., by a mem-initializer)--there is also no similar compiler-generated initialization
    d. a delegate base's virtual functions are not considered when trying to determine the functions being overriden in a derived 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
```

[15] 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 a delegate base – 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 a delegate base adds to the conversion-operator is for unified-function-call - where name lookup for uniform call syntax would consider the conversion.

5. Delegate bases can be public, private or protected, just like regular bases.

6. A type can inherit a delegate base that itself inherits from a delegate base mingled with regular bases within a class lattice. For each 'delegate base' relationship a conversion to delegate-base can designate a subobject.

7. A delegate base does not need to be complete in a TU, unless name lookup within the class or after its definition, requires delving into the delegate (i.e., name is not found in the inheriting 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 delegate base in one case and find the name in a namespace scope in the other).

8. OK for a delegate-base to be a "final" class

9. Using-member declarations from delegate bases work

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

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

12. A class that has a conversion to itself can not be a delegate base; base conversion operators are inherited.

13. A pointer to member of a delegate base can not be converted to a derived pointer.

```
struct A { int x; };  struct B : using A { using A::x; };
int A::*px = &B::x; // OK
int B::*px2 = &B::x; // NOT ok
```

4 Pros and Cons

Advantages of Delegate Bases over N4477 Operator Dot (R2):

- ➔ Three composable, non-monolithic sub-features with simple idiomatic interfaces as opposed to one operator with wide-ranging consequences.
- ➔ using DelegateBase::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 or resolve data-member ambiguities.
- ➔ Does not complicate member lookup further
- ➔ Does not require empty-base optimization when delegate inheritance 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.
- ➔ The process of member access, we feel, is simpler with delegate bases. Consider the process for delegate bases where member lookup and overload resolution occurs twice (see Appendix A for a deeper look): first when looking up the member name and second when finding the right member conversion operator from the derived object type (object-expression) to the delegate base source type (based on the value category of the object expression). Only this second step has to occur recursively if the hierachy contains delegate bases that have their own delegate bases, and if a unique path from the source to the target does not exist, an ambiguity occurs. **Now consider for N4477 where something more exotic probably has to occur** to model the behavior as we understand it (assuming we do, and it is not unlikely that we actually do since some of the subtleties did not seem obvious to us from N4477 and the wording is still being worked on, so our apologies to their authors if we misrepresent, and welcome any clarifications or simplifications): first member lookup only into the static type of the object expression occurs

(not delving into the bases yet) - if the name is found, stop and perform overload resolution. If a name is not found, then perform lookup into all the bases and also lookup into all the return types of all overloaded operator-dots within (i.e. including bases) the entire class hierarchy and the class hierarchy of all the types returned by any of the operator-dots, recursively, while tracking the path (sequence of operator-dots followed) to every name/member declaration. If you end up with a declaration from a regular base and also through lookup through an operator-dot, an ambiguity occurs. If the declaration-set only contains declarations from Base subobjects, follow 10.2's (not-simple) rules for ambiguity followed by routine overload resolution followed by the ambiguity check for base class conversions. Otherwise perform overload resolution (we're not clear whether the value-category of the object-expression or the return type of the operator-dot is used during this step. It is also worth noting that it is unclear to us how the ambiguity checks for 10.2 – designed mainly to deal with virtual bases and conflicting declarations from different bases at the same depth – would need to be tuned to account for the same declarations entering through different operator-dots at different depths) amongst all found declarations. If overload resolution succeeds, then invoke the tracked sequence of operator-dots associated with that declaration, starting with the object-expression (i.e the left hand side of the member access) to get the eventual object on which to access the member. If a unique such path does not exist, we suspect an ambiguity error occurs.

Advantages of N4477 Operator Dot (R2) over Delegate Bases:

➔ 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) - delegate bases obviously require the types that are being inherited to be defined outside the class doing the inheriting.

Neutral:

➔ Delegate inheritance 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 delegate inheritance; 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.

5 Implementation

An implementation for the auto-alias based on N4035 was implemented by one of the authors in 2014: https://github.com/faisalv/clang/commits/using-auto-conv

An implementation using Clang for Delegate Bases and their conversion operators is being worked on.

6 Teachability

We expect this to be relatively easy to teach to programmers who are familiar enough with inheritance, composition and user-defined conversions - since delegate inheritance 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 may be taken: with delegate inheritance, the derived class no longer owns a base-class instance, but instead borrows its interface; this interface-borrowing becomes the core aspect of inheritance. 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.

7 Core Wording

We shall await feedback from EWG before presenting core wording that would be appropriate for a CWG audience.

8 Future Directions

1. **Sizeof Alias : using sizeof = <type>**
   An example in section 6.1 of N4477 suggests that taking the sizeof a class containing an overloaded operator-dot should compute the size of the return type of the operator-dot. We see no other mention of this functionality in the paper. If this is truly desired functionality, we propose a sizeof alias as above. If a type contains a sizeof alias, sizeof applied to that type would always compute the size of the aliased type. We might also need to provide a way to bypass the sizeof alias.

2. **Lifting Restrictions:**
   Removing any or all of the non-fundamental restrictions mentioned in section 3.

3. **Explicit delegate-base 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

4. **Generation of default conversion operator to delegate bases:**
   We could agree upon a convention so that the compiler generates all the base conversion operators - For e.g., if there is one data member of pointer or reference or object type of a delegating base, automagically 'default' base conversion functions get generated that returns it.

5. **Virtual Delegate Bases:**
   Presently we give no semantics to virtual delegate bases, but these could be considered a promise from the user that conversion to the virtual base may be achieved by arbitrarily choosing a path in the inheritance graph.

6. Allowing deleted base conversion operators to disown bases (delegate or otherwise) from unwanted name lookup and conversion.

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

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

9 Acknowledgment/References

N4477, “Operator Dot (R2)”, Stroustrup, Dos Reis

N4035, "Implicit Evaluation of auto Variables and Arguments", Falcou, Gottschling, Sutter

The authors would like to also 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 a Delegate Base: A Deeper Look

```
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 delegate 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
                derived-to-delegate-base 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 delegate base types to convert to is determined by the inheritance 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 inherits by
delegation from the type needed (call it D [in our specific example 'Wrap<T>' directly inherits
from 'T']); overload resolution for the base 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
```

Appendix B: Examples from N4477 and their Equivalents using P0352

N4477 [4.5] Recursive Operator Dot Concerns

```
template<class X>
class Ref {
public:
  explicit Ref(int a) :p{new X{a}} {}
  X& operator.() { /* … */ return *p; }
  // clone the value: (&(a.operator.()))->clone()
  Ref(const Ref& a) { p = (&a)->clone(); }
  Ref(Ref&& a) : p{(&a)->p} { (&a)->p=nullptr; }
// …
private:
  X* p;
};
```

Current Proposal

```
template<class X>
class Ref : public using X {
  operator X&() { /* … */ return *p; }
public:
  explicit Ref(int a) :p{new X{a}} {}

  // clone the value: (a.operator X&()).clone()
  Ref(const Ref& a) { p = a.clone(); }
  Ref(Ref&& a) : p{a.p} { a.p=nullptr; }
// …
private:
  X* p;
};
```

N4477 [4.7]

```
struct S {
  int& operator.() { return a; }
  int a;
};


S s {7};
int x = s.operator.(); // x = s.a
s.operator.() = 9; // s.a = 9
```

Current Proposal

```
struct S : using int {
  int a;
  int operator=(int);
private:
  operator int&() { return a; };
};

S s{7};
int x = s;  // OK
int &x = s; // Error: can not bind directly to reference
s = 9; // uses S::operator=(int)
```

N4477 [4.9] Const Overloading

```
struct T { void f(); void f() const; };

struct SS {
  T& operator.() { return *p; }
  const T& operator.() const
  { return *static_cast<const T*>(p); }
  // ...
private:
  T* p;
}

void (SS& a, const SS& ca)
{
  a.f(); // calls non-const member T::f()
  ca.f(); // calls const member T::f()
}
```

Current Proposal

```
struct T { void f(); void f() const; };

struct SS : using T {
  operator T&() { return *p; }
  operator const T& () const
  { return *static_cast<const T*>(p); }
  // ...
private:
  T* p;
}

void (SS& a, const SS& ca)
{
  a.f(); // calls non-const member T::f()
  ca.f(); // calls const member T::f()
}
```

N4477 [4.9][16] Model Multiple Inheritance

```
struct T1 {
  void f1();
  void f(int);
  void g1(int);
  void g();
  int m1;
  int m;
  int n;
};

struct T2 {
  void f2();
  void f(const string&);
  void g1(const string&);
  void g();
  int m2;
  int m;
  int n;
};

struct S3 {
  // use if the name after . is a member of T1
  T1& operator.() { return p; }

  // use if the name after . is a member of T2
  T2& operator.() { return q; }









  // ...
private:
  T1& p;
  T2& q;
};

void (S3& a)
{
  a.g();  // error: overload-resolution failure btw
          // T1::g() & T2::g(), not a lookup ambiguity.
  a.f1(); // calls a.p.f1()
  a.f2(); // call a.q.f2()
  a.f(0); // calls a.p.f(0)
  a.f("asdf"); // call a.q.f string("asdf")
  a.g1(0);     // OK, calls a.q.g1(0)
  auto x0 = a.m; // error: ambiguous
  auto x1 = a.m1; // a.p.m1
  auto x2 = a.m2; // a.q.m2
  auto x3 = a.n;  // error: ambiguous
}
```

Current Proposal

```
struct T1 {
  void f1();
  void f(int);
  void g1(int);
  void g();
  int m1;
  int m;
  int n;
};

struct T2 {
  void f2();
  void f(const string&);
  void g1(const string&);
  void g();
  int m2;
  int m;
  int n;
};

struct S3 : using T1, using T2 {
  // use if the name is found in T1
  operator T1&() { return p; }

  // use if the name is found in T2
  operator T2&() { return q; }

  // We need using declarations to merge overload sets
  // from different bases at the same depth, as usual.
  using T1::f; using T2::f;
  // NOTE: we do not do using-declarations for 'g1'
  // which results in a member lookup ambiguity in
  // today's C++ for 'g1' and in our proposal,
  // but NOT with N4477.

  using T2::n; // Disambiguate, not possible in N4477
  // ...
private:
  T1& p;
  T2& q;
};

void (S3& a)
{
  a.g();  // error: usual mem-lookup ambiguity (10.2 /6.2)
          // This never gets to overload-resolution.
  a.f1(); // calls a.p.f1()
  a.f2(); // call a.q.f2()
  a.f(0); // calls a.p.f(0)
  a.f("asdf"); // call a.q.f string("asdf")
  a.g1(0); // error: ambiguous, usual member lookup (10.2)
  auto x0 = a.m; // error: ambiguous
  auto x1 = a.m1; // a.p.m1
  auto x2 = a.m2; // a.q.m2
  auto x3 = a.n; // OK: as usual, because of using-decl
}
```

[16] Example is slightly augmented

N4477 [4.8]

```
template <class X> class Ref {
public:
  struct Wrap {
     Wrap(X* pp) : p{pp} { before(); }
     ~Wrap() { after(); }
     X& operator.() { access(p); return *p; }
     X* p;
  };
  Ref(X* pp) :p{pp} {}
  Wrap operator.() { return {p}; }
  // …
private:
  X* p;
};

void foo(Ref<X>& x )
{
  x.foo(); // x.operator.().foo()
  // => Wrap(x.p).foo()
  // => Wrap(x.p).operator.().foo()
  // => before(); access(x.p); (x.p)->foo(); after()
  auto v = x.bar(); // auto v = x.operator.().bar();
  // auto v = Wrap(x.p).bar();
  // roughly: before(); access(x.p);
  //    auto v = (x.p)->bar(); after()
}
```

Current Proposal

```
template<class X> class Wrap : public using X {
  Wrap(X *pp) : p{pp} { before(); }
  ~Wrap() { after(); }
  operator X&() { access(p); return *p; }
  X *p;
};

template<class X> class Ref : public using Wrap<X> {
public:
  Ref(X *pp)      : p{pp} {}
  Ref<X>& operator=(X& r)  { *pp = r; return *this; }
  // …
private:
  X* p;
  operator Wrap<X>() { return {p}; }
};

void foo(Ref<X>& x )
{
  x.foo(); // x.operator Wrap().operator X&().foo()
  // => before(); access(x.p); (x.p)->foo(); after()
  auto v = x.bar();
  // auto v = Wrap(x.p).bar();
  // roughly: before(); access(x.p);
  //    auto v = (x.p)->bar(); after()
}
```

N4477 [6.1] Pimpl

```
//// header:
class X {
public:
  int foo(); // noninline => rather stable
private:
  int data; // not used through operator.()
};

template<class T>
class Handle {
public:
  // access the T exclusively through p
  Handle(T* pp) :p{ pp } {}
  T& operator.() { return *p; } // don’t leak p
private:
  T* p;
};

//// TU1:

// uses representation: not very stable
int X::foo() { return data; }

//// TU2:
void f(Handle<X> h)
{
  int d = h.foo();
}

int main()
{
  Handle<X> hx{ new X{ 7 } };
  f(hx);
  sizeof(hx); // size of X (not size of Handle<X>)
              // because of operator-dot overload.
}
```

Current Proposal

```
//// header:
class X {
public:
  int foo(); // noninline => rather stable
private:
  int data; // not used through operator.()
};

template<class T>
class Handle : using public T {
public:
  // access the T exclusively through p
  Handle(T* pp) :p{ pp } {}
  operator T&() { return *p; } // don’t leak p
  using sizeof = T;
private:
  T* p;
};

//// TU1:

// uses representation: not very stable
int X::foo() { return data; }

//// TU2:
void f(Handle<X> h)
{
  int d = h.foo();
}

int main()
{
  Handle<X> hx{ new X{ 7 } };
  f(hx);
  sizeof(hx); // size of X (not sizeof Handle<X>)
              // because of 'using sizeof' extension
}
```

N4477 [6.2] Adding Operations To a Proxy

```
template<typename T> // T must have a <
struct totally_ordered {
  totally_ordered(const T& t) : val{t} { }

  // NOTE: Modified original example in N4477 to
  // use arrow-operator to avoid any confusion
  // regarding operator-dot recursion

  bool operator<=(const totally_ordered& y) const
  { return not( (&y)->val < this->val); }

  bool operator>(const totally_ordered& y) const
  { return (&y)->val < this->val; }

  bool operator>=(const totally_ordered& y) const
  { return not (this->val < (&y)->val); }

  // don’t leak a pointer to val
  T& operator.() { return this->val; }

private:
  T val;
  // here is the value (totally_ordered is not a
  // reference type)
};

struct basic_ordinal {
  basic_ordinal(std::size_t i) : val{i} { }

  bool operator<(basoc_ordinal y) const
  { return val < y.val; }
private:
  std::size_t val;
};

using Ordinal = totally_ordered<basic_ordinal>;
```

Current Proposal

```
template<typename T> // T must have a <
struct totally_ordered : using T {
  totally_ordered(const T& t) : val{t} { }

  bool operator<=(const totally_ordered& y) const
  { return not( y.val < val); }

  bool operator>(const totally_ordered& y) const
  { return y.val < val; }

  bool operator>=(const totally_ordered& y) const
  { return not (val < y.val); }

private:
  // don’t leak a pointer to val
  operator T&() { return val; }
  T val;
  // here is the value (totally_ordered is not a
  // reference type)
};

struct basic_ordinal {
  basic_ordinal(std::size_t i) : val{i} { }

  bool operator<(basoc_ordinal y) const
  { return val < y.val; }
private:
  std::size_t val;
};

using Ordinal = totally_ordered<basic_ordinal>;
```

N4477 [6.3] Remote Object Proxy

```
template<class T>
class Cached {
public:
  Cached(const string& n); // read n into memory and bind
                           // to obj
  ~Cached(); // write obj back into s (transaction safe)
  void flush(); // write obj back into s (transaction safe)
  void read(); // read name into obj
  // ...
  T& operator.() { if (!available) read(); return obj; }
private:
  T& obj;
  bool available; // a local copy is available through obj
  string name;
};
```

Current Proposal

```
template<class T>
class Cached : using public T {
public:
  Cached(const string& n); // read n into memory and bind
                           // to obj
  ~Cached(); // write obj back into s (transaction safe)
  void flush(); // write obj back into s (transaction safe)
  void read(); // read name into obj
  // ...
private:
  operator T&() { if (!available) read(); return obj; }
  T& obj;
  bool available; // a local copy is available through obj
  string name;
};
```

N4477 [6.4] Optional

```
template<typename T>
class Optional {
// NOTE: The example lifted from N4477 might need to
// be adjusted to use an explicit 'this' and arrow-operator
// to aid with clarity about recursive op-dot calls.
public:
  T& operator.()
     { if (opt_empty()) throw Empty_optional{}; return obj;
}

  Optional() : dummy{true}, b{false} { }
  Optional(T&& xx) : obj{xx}, b{true} { }
  // ...
  Optional& operator=(const T& x) {
    if (opt_empty()) new(&obj) T{x};
    else obj=x;
    b=true; return *this;
  }
  // ...
  bool opt_empty() { return !b; }

  T operator T()  {
    if (opt_empty()) throw Empty_optional{};
    return obj;
  }

  T value_or(T&& v) { return (opt_empty()) ? v : obj; }

  template<typename Fct>
  T value_else(Fct err)
    { return (opt_empty()) ? err() : obj; }
private:
  union {
    T obj; // only valid/initialized if b==true
    bool dummy; // used only to suppress init of obj
  };
  bool b;
};

Optional<complex<double>> oz0 {};
Optional<complex<double>> oz2 {{1,2}};

Complex<double>>& r0 = oz0; // error: the result of
                                  // oz0.operator.() must be
used.
auto z0 = oz0; // throws
auto x1 = oz2; // x1 is complex<double>; x1 == {1,2}

if (!oz0.opt_empty()) {
   // use oz0
}

auto x2 = oz2.value_or({0,0});
oz0 = {3,4};
auto x3 = oz0 * oz1 + {5,6}; // x3 is complex<double>
auto x4 = oz1.value_else([] {
                cerr << "Hell is loose!";
                return complex<double>{0,0};
          });
```

Current Proposal

```
template<typename T>
class Optional : public using T {



  operator T&()
     { if (opt_empty()) throw Empty_optional{}; return obj;
}
public:
  Optional() : dummy{true}, b{false} { }
  Optional(T&& xx) : obj{xx}, b{true} { }
  // ...
  Optional& operator=(const T& x) {
    if (opt_empty()) new(&obj) T{x};
    else obj=x;
    b=true; return *this;
  }
  // ...
  bool opt_empty() { return !b; }

~~  T operator T()  {~~
~~    if (opt_empty()) throw Empty_optional{};~~
~~    return obj;~~
~~  }~~
  using auto = T;
  T value_or(T&& v) { return (opt_empty()) ? v : obj; }

  template<typename Fct>
  T value_else(Fct err)
    { return (opt_empty()) ? err() : obj; }
private:
  union {
    T obj; // only valid/initialized if b==true
    bool dummy; // used only to suppress init of obj
  };
  bool b;
};

Optional<complex<double>> oz0 {};
Optional<complex<double>> oz2 {{1,2}};

Complex<double>>& r0 = oz0;// error: conversion op is
                                  // private for direct-ref
binding
auto z0 = oz0; // throws
auto x1 = oz2; // x1 is complex<double>; x1 == {1,2}

if (!oz0.opt_empty()) {
   // use oz0
}

auto x2 = oz2.value_or({0,0});
oz0 = {3,4};
auto x3 = oz0 * oz1 + {5,6}; // x3 is complex<double>
auto x4 = oz1.value_else([] {
                cerr << "Hell is loose!";
                return complex<double>{0,0};
          });
```