Implicit user-defined conversion functions as operator.()

Document #:	P3298R0
Date:	2024-05-22
Project:	Programming Language C++
Audience:	EWGI

Reply-to:	Bengt Gustafsson
	<bengt.gustafsson@beamways.com>

Contents

1 Abstract

By tagging user-defined conversion functions with an `implicit` specifier the conversion rules change to match that its class *inherited* the unreferenced return type of the `implicit` user-defined conversion function (ICF). This allows implementing "smart references" and similar which was the target application for the user-defined `operator.()` proposals. The name lookup rules for an ICF is the same as for inheritance so there is little special semantics not covered by inheritance related rules.

Apart from the use for `operator.()` the correspondence to inheritance allows the object to be provided as the argument when the parameter type of a function is constructible from the type returned from the ICF. This possibility is available in the more recent `operator.()` proposals too but it is less obvious that this is the case if the feature is spelled `operator.()`.

The context specific keyword `implicit` may be considered confusing as in some texts a conversion function *not* marked as `explicit` is called an implicit conversion function. This proposal regards conversion functions without explicit as "normal" conversion functions with conversion rules somewhere between implicit and explicit conversion functions. Bike shedding may be done for the specifier keyword and luckily there is no need to avoid keywords that may have been used as regular identifiers as it is contextual and only a keyword if followed by the full keyword `operator`. However, `implicit` seems like a good choice as it is intuitively understood as a conversion that is more eagerly employed than a "normal" conversion function.

2 Previous work

A lot of effort has been put into bringing `operator.()` into C++. The latest seems to be [P0416R1] from 2016. An earlier version of this, [N4477] is discussed among other efforts here: background. These papers use the terms *handle* and *value* to refer to the type containing the `operator.()` and the type returned by `operator.()`, respectively. For clarity the same terms are used here.

The idea of modeling the feature after inheritance was presented earlier in 2016 in [P0352R0] and if that paper had gone forward this proposal would not have been needed. Most of this paper was written without being aware of [P0352R0] and is published to push this feature forward and as some aspects are different and more consistent than previous proposals.

The essential features of all three approaches are the same since [P0416R1] removed some exotic aspects of [N4477] such as sizeof returning the size of the value. The differences are mainly syntactical and pertain to how the feature is presented and thus mostly relate to teachability:

P0461R1 allows creating user defined `operator.()` functions, but these are called in places where the built in operator. is not called, i.e. when the value type is needed as a function argument type. This makes the formulation as an operator.() less logical.

P0352R0 allows a `using` keyword before a type in the base class list to indicate what it calls *delegate inheritance.* Delegate inheritance *requires* a user defined conversion function to be declared. This double book-keeping is rather confusing especially considering that such delegate inheritance is allowed for fundamental types.

This proposal uses an `implicit` specifier on a conversion function to indicate that it can be applied in the same circumstances as a sub class to base class conversion, and with the same conversion cost. As these circumstances include when a dot is applied an ICF works the same as a user defined `operator.()` in this case. Likewise, if an `operator->` returns a handle the value type is considered if the handle type does not have the name specified on the rhs.

Other situations where value classes are considered includes implicit upcasts when used as function arguments (including operators) and applying the scope resolution operator to access names in the value type.

2.1 Comparison to P0416R1

[P0416R1] defines operator.() to be called whenever a dot is placed after an object of the handle type, unless the name after the dot is found in the handle class. More surprisingly it also calls `operator.()` when an operator is applied to the handle but is defined only for the value . The `operator.()` is also called when a handle is used as argument to a function requiring a value, which may seem even more surprising, although operators are just functions anyway.

In [P0416R1] (section 4.15) it is explicitly stated that the scope resolution operator can not find names in types returned by `operator.()` which seems to hamper using handle classes in lieu of the corresponding value classes.

```
class Value {
    using Type = int;
    void f();
    static void g();
    Value& operator++();
};

Value& operator--(Value&);

void f(Value& a);

struct Handle {
    Value& operator.();
};

Handle pr;

pr.f();     // Obviously works as there is a dot there.
++pr;       // Works according to P0416R1
--pr;       // Works according to P0416R1
f(pr);      // Works according to P0416R1

Handle::Type x;     // Does not work.
Handle::g();        // Does not work.
```

When piggy-backing on inheritance for name lookup as this proposal does all of these operations trivially work and is understood by inspecting what happens if `Handle` had inherited from `Value`.

2.2 Comparison to P0352R0

The main difference is that this proposal uses a specifier on the declaration of the conversion function itself to indicate its special name lookup rules, while [P0352R0] used a regular conversion function in combination with a special inheritance declaration `class Handle : using public Value`. This double book-keeping is an extra source of inconsistency and opens for questionable rules for what happens if the inheritance and conversion function have different access control and how overloads of the conversion function with different cvref qualifiers are motivated.

The artificial limitation of [P0352R0] that a nested class can not be returned from a conversion function which is caused by the mentioning of the nested class in the base list is not present with this proposal as the return type is only mentioned in the declaration of the conversion function.

2.3 Relation to N4035

The proposal [N4035] regarding `using auto = T` as a means of defining what type an object which can be converted to some other type should deduce to. [P0352R0] uses this to optionally get deduction to the value type.

This proposal could be combined with [N4035] in the same way if both get standardized. Doing so is sometimes important to avoid dangling references. See for instance the *string interpolation result* example below.

2.4 Relation to P0700R0

Hopefully this paper will be more clear than [P0352R0] so that the questions in [P0700R0] can be properly addressed. The main reason for modeling name lookup from inheritance is that it avoids new rules and leaves other aspects of the handle class up to the programmer, especially if [N4035] gets adopted.

3 Introductory example

Here is a simple example of a `Proxy` class that transparently works as its `T` class when it comes to overload resolution. As the implicit conversion is applied in the same situations as a base class would be looked up this effectively works as an `operator.()`. `Proxy<T>` objects can for instance be stored in a `vector` and the vector elements can be accessed via `vec[i].member` despite the fact that the vector elements are actually just pointers to objects located somewhere else.

```
template<typename T> class Proxy {
    Proxy(T& object) : m_ptr(&object) {}

    implicit operator T&() { return *m_ptr; }
    implicit operator const T&() const { return *m_ptr; }

private:
    T* m_ptr;
};

struct MyClass {
    using Type = int;
    int x;
    void f();
    static void s();
};

void g(MyClass& o);

void test()
{
    MyClass obj;
    Proxy<MyClass> p(obj);
    Proxy<MyClass>* pp = &p;

    p.f();          // As Proxy<T> does not have an f check its bases and ICF return types
    p.x = 43;       // As Proxy<T> does not have an x check its bases and ICF return types
    g(p);           // As g does not take a Proxy<T> check its bases and ICF return types
```

```
    // All name lookup considers names in bases and ICF return types
    Proxy<MyClass>::Type anInt;

    // operator-> considers names in bases and ICF return types
    pp->f();
    pp->MyClass::f();  // Redundant but allowed and useful if Proxy had had its own f.

    Proxy<MyClass>::s();    // Call static method of MyClass using name lookup.
}
```

Proxy works like `std::reference_wrapper` but you can use `.` instead of `->`. and it can be used as an argument without dereferencing with `*`. This works exactly as if `Proxy<T>` had inherited `T` when it comes to name lookup and function overload resolution.

Even if there are multiple ICFs in the handle class inheritance name lookup rules apply: This works exactly as multiple inheritance. Even if the value type has ICFs we're covered, it works exactly as if the base class has a base class.

An ICF can be combined with a non-explicit constructor or user defined conversion function as it counts as a base class conversion for purposes of overload resolution, this makes the following example legal:

```
class A {
};

class B {
    B(const A&);
};

void f(B b);

A a;
Proxy<A> pa(a);

f(pa);       // This works despite invoking both ICF on pa and constructor of B.
```

4 Overload resolution rule implications

More formally an ICF has the rank of *conversion*, just like a derived to base conversion.

An ICF constitutes a *standard conversion* which means that it can be combined with a *user-defined conversion* i.e. a constructor call or non-implicit user-defined conversion function. It can also be combined with derived to base class conversions or further ICFs just like multi-level derived to base conversions are possible. There may be an entire sequence of combined ICFs and derived to base conversions in any order before *and* after a constructor or non-implicit user defined conversion function.

If the conversion sequence that the ICFs are part of is not distinguishable by rank and the rules for const and reference binding does not break the tie the rule of fewest combined number of derived to direct base and implicit conversion steps is used to break the tie. If multiple such paths have the same number of steps the conversions are equally good and may render the call ambiguous (if other argument conversions don't break the tie).

4.1 Pointer conversion rules

A pointer to derived can be converted to a pointer to base, which indicates that a pointer to a handle object should be possible to convert to a pointer to the value type.

This should at least work if the user-defined conversion returns a reference, under the assumption that this reference will refer to some object with sufficient lifetime. However, unlike inheritance, there is no guarantee that this is true provided by the language.

If the ICF returns by value the pointer would point to a temporary which likely has shorter lifetime than the pointer. This should not be allowed as it would invite dangling pointer problems. This is thus proposed to be illegal.

4.2 Returning a base type

It is not very useful to return a base type from an ICF in a derived type. By the rules of inheritance the names in the base are always ambiguous in this case (corresponding to the same class inherited twice in the base class hierarchy). One possible use would be to make a base further up in the hierarchy with protected inheritance publicly available. But who uses protected inheritance anyway?

```
class B {
    void f();
};

class S : public B {
    implicit operator B&() { return other_b; }

private:
    B other_b;
};

S s;
s.f();      // Ambiguous.
```

To make this somewhat more useful one idea would be to bend the rules so that the base instance returned by the ICF is preferred, which would allow using an ICF to specify a preferred instance of the base class in cases where it is currently ambiguous. This does not seem to be an important enough use case to deviate from the rule that name lookup works as for inheritance, so it is not proposed.

4.3 Returning a derived type

When returning a derived type from an ICF of a base type the names in the base type itself are considered before names in the ICF's return type are considered. When the ICF is considered and further derived to base conversions end up back at the class declaring the ICF compilers must make sure not to try the same ICF again as it could end up as an infinite recursion or loop during compilation. This can easily be avoided, but it would also be possible to forbid implicit conversion operators to a derived class. As the compiler logic to avoid the infinite recursion or issue the error message is the same it is proposed to allow this construct, mandating the compilers to block the meaningless repeat of the same ICF.

```
class S;

class B {
    implicit operator S&();     // No error here
};

class S : public B {            // No error here
};

S s;
s.f();      // Error: No f found. Compiler is not allowed to crash or hang.
```

4.4 Qualified name lookup and the scope resolution operator

To explicitly access a member of a base class you can qualify its name with the base class name. In this case the compiler finds the base class in the inheritance tree from the declared type of the object or pointer the name is looked up in. In this case ICF return types are included in this search in the same way as base classes.

When applying the scope resolution operator to a handle type the names in value types returned by its ICFs are considered in concert with names in base classes of handle. This allows all the uses of such names, including nested types, enumerators, type aliases, static member functions and static data members.

5 implicit is not a full keyword

`implicit` is a contextual keyword, you can still use it as an identifier, it only has the special meaning when followed by the keyword `operator`. This type of prefix context-sensitive keywords were pioneered by Corentin Jabot in the `universal template` keyword combination suggested for universal template parameters. There is no valid syntax where an identifier can be followed by the keyword `operator`.

6 Other aspects to consider

The correspondence to inheritance is limited to name lookup rules. Other aspects are not the same as for inheritance, where implicit conversion operators instead work as their non-implicit counterparts.

Most of these aspects have been covered in previous proposals but the coverage of member pointers, virtual ICFs etc. here is more detailed, with rationale for why different possibilities are included or excluded.

6.1 An implicit cast operator can return any type

It is possible to return a non-class type from an implicit user-defined conversion function. It would be close at hand to forbid this as you can't inherit from fundamental types, pointers or C-arrays. Also, as these types don't have members the resulting `operator.()` functionality would not come to use. There would be other uses though, for instance we could have a convenience class like this:

```
template<typename T> class Stringable {
public:
    Stringable() = default;
    Stringable(const T& src) : m_object(src) {}

    Stringable& operator=(const T& src) { m_object = src; return *this; }

    implicit operator T&() { return m_object; }
    implicit operator const T&() const { return m_object; }

    std::string to_string() const { return std::to_string(m_object); }
    [[nodiscard]] bool from_string(std::string_view str);

private:
    T m_object;
};


void test()
{
    Stringable<int> a = 1, b = 2;

    auto str = a.to_string();           // Ok
```

```
    auto str2 = (a + b).to_string();    // Nope: a + b adds the int "bases" returning an int!
}
```

Now we must imagine that `int` is a base class of `Stringable<int>` and thus we see that as no `operator+()` exists for `Stringable<int>` we can instead call the "base class" `operator+()`, which is just int addition.

6.2 Final classes

If we model ICFs too closely after inheritance it would be logical to disallow returning final classes. However, this is probably not a good idea, `final` is about optimizing virtual calls and other reasons to avoid inheriting further. These reasons for `final` don't apply to value classes returned by ICFs.

6.3 Overriding virtual methods

Allowing overrides of methods of the value type in the handle type is not possible as that would require modification of the vtable pointer in the value object, which is potentially not owned by the Handle.

It is allowed to declare what looks like an override of a virtual function of the value class in the handle class. This is however a new function, so It is prohibited to use the `override` specifier. This provides some measure of protection in case programmers think that such overriding is possible.

6.4 Virtual inheritance

A very odd and unusual case would be if both the Handle and the Value inherit the same type virtually.

```
class Base {
    void f();
};

class Sub : public virtual Base {};

class Proxy : public virtual Base {     // Error in case 1.
    implicit operator Sub&();           // Error in case 2.
};

Proxy p;
p.f();      // Error in case 3, ambiguous call.
```

This causes the same problem as with virtual functions, that the vtable or other data structure used by the ABI to find the virtual base would not be possible to change in the value object referred to by the handle object. The remaining question is whether it should be prohibited to:

1. Inherit anything virtually if the class has an ICF.
2. Declare an ICF with Value type that inherits the same type virtually as the Handle type containing the ICF.
3. Try to access the virtual base's members from the type containing the ICF function in this case, i.e. make the access ambiguous despite that it would be allowed in the corresponding inheritance case.

As this is a very odd case it would be fine with any of these options, but 1 seems like a rather odd restriction. 2 has some logic to it, but 3 is consistent with the underlying reality: There are two different Base sub-objects and no way to select which one is better except using the scope resolution operator which in this case can only be used to select the base of `Sub`, not the base of `Proxy`.

6.5 Member pointers

Member pointers and applying them to objects with ICFs requires extra handling. Remember that member pointers follow the contra-variant principle: The member pointer must be for a member of a base class (or

the same class) as the object it is applied to. When the member pointer is created using the `&Class::member` syntax the compiler must check in which base class of `Class` the `member` is found. To get from the `Class` object reference that is provided at the call site to the object reference of the class where `member` was found may today require offset adjustments and/or virtual base class pointer indirections. The only way to implement this is to pack more information into the member pointer. As explained here thunks can't be used as it makes casting to a base class pointer type impossible with multiple and/or virtual inheritance. Compilers have to inspect the base classes to see if there is any multiple inheritance and/or virtual inheritance, and decide on a suitable layout of a member pointer type for this class. With ICFs the member pointer type must have room for pointers to as many ICF functions as maximally needed to get to any class the class can be converted to. In fact unbounded member pointer size is already an issue if there are multiple levels of virtual inheritance. None of the major compilers can handle this today (right?).

Note that it is the conversion of an object reference to the class for which the member pointer is created to the class where the member is found that has to be encoded in the member pointer, the conversion from the pointer/reference type to the declared class of the member pointer is implemented at the point where the member pointer is used and is fully known there.

A new variant of function pointer layout can be devised that has an option for calling an ICF instead of doing an indirection to a virtual base. This is not an ABI breaking change as only classes with ICFs somewhere in the inheritance hierarchy would need the new member pointer layout, and there are no such classes in old code.

```
class Value {
    void f();
};

class Handle {
    void g();
    implicit operator Value&();
};

using HPP = void(Handle::*)();
HPP pf = Handle::f;
HPP pg = Handle::g;

Handle h;
(h.*pf)();      // ICF must be called
(h.*pg)();      // ICF must not be called.
```

As seen in this example the function pointer type `HPP` must contain information about whether the ICF must be called to convert a `Handle` reference to a suitable object reference to use when calling the function pointed to by the `HPP`. It is also quite obvious that this variety of function pointer layout must be used whenever an ICF is found anywhere in the inheritance hierarchy of Handle, as it is possible to assign a member pointer type to a member of any base class. It should also be obvious that no old code compiled before ICFs were a thing can use a member pointer to a member of a class which has a ICF, or as a corollary, that it is not allowed to add an ICF in a class hierarchy if any subclass of that class is used by code that is not feasible to recompile. This is no stranger than the fact that you can't add a member to class if you can't recompile code that uses it.

Thus, while there is no ABI break involved there is certainly ABI design. Currently ABIs are designed so that one virtual inheritance level and one or two non-first inheritances (i.e. object pointer offset additions) can be represented. With ICFs in the mix it may be appropriate to design a more complete solution which covers all possibilities. This may make member pointers extremely large but the layout can always be specified knowing the contents of the inheritance hierarchy of the class of the member pointer type (`Handle` in the example above). An ABI designer may also choose to add an indirection level for more complex cases, for instance limiting the size of the pointer to two regular pointers, and if more is required let the second pointer point at a descriptor block.

Member pointers that point to non-virtual member functions have to contain the address of the function to call

in addition to any information about how to modify the object reference before doing so. Thus the minimum size of a member pointer must be two memory pointers, one to the function and one to the ICF. Some low bits may be used to differentiate different modes, or a conversion function created for the purpose could always be called. Many options are available to ABI designers and compiler vendors, as they already limit the complexity of scenarios where member pointers are allowed. Note that if the ICF is virtual it needs a different encoding where the address of the ICF is replaced by a vtable offset.

Note that if D3312R0 for *overload-set-types* is adopted taking the address of an overloaded member function would be allowed. However this is a compiler time facility so it does not change anything regarding member pointers, except when they can be created. The *overload-set-type* is created by `&Class::member` but if member is an overloaded function nothing more happens until it is statically cast to a member pointer type, at which point the same member pointer layout selection is made as if the `static_cast` was applied to the overloaded member function name directly.

6.6 Explicitly calling virtual methods in a value class

If both the Handle and the Value have declared the same member name it would be tempting to use `handle.Value::name` as a way to reach the Value's version of the name. This is proposed. However, if `name` is a virtual function we are in a pickle. With inheritance we use `Value::name` to call the implementation of the virtual method as seen from `Value`. This is usually not what you want in the case of name clashes between the Handle and the Value, but that's what the rules say. If we change those rules there would be no way to call the exact function specified, and it would be confusing if it worked differently than if Value was a base class of Handle.

To retain virtual dispatch you instead write`static_cast<Value&>(handle).name(args)` which forces the implicit conversion function to run (just as if it hadn't been implicit). Then virtual dispatch is performed as any like-named member function in the Handle is no longer hiding the target function, and no scope resolution operator was used to pin down which override to call.

This said named members of generic Handle classes should be avoided as there is always a risk of clashes, and users of the handle class don't want to `static_cast` all the time just to make sure the Handle's members are not interfering.

6.7 Returning incomplete classes

While it is not possible to inherit from an incomplete class it should be possible to define an ICF that returns an incomplete type, this is consistent with any other function. Even when returning by value it is possible to *declare* an ICF returning an incomplete type, but not defining it.

Using the returned value is restricted as usual, which means that failing name lookup in the handle class is an error if the return type of any ICF is incomplete. *This is true even if the Handle has other ICFs returning complete types or has base classes, as the incomplete type prevents checking for ambiguous name lookup.*

Using `static_cast` to an incomplete Value type is allowed but the returned type is still incomplete and has the same restrictions.

6.8 Downcasting

For inheritance you can downcast a reference using `static_cast` if you know what you're doing and with `dynamic_cast` if you don't. Performing `static_cast` from a class to an object returning that class from an ICF is impossible as the address offset between them is not a constant. This is consistent with the fact that `static_cast` can not downcast from a virtual base to a subclass for the same reason.

It is possible to downcast to a derived class inheriting the base class virtually using `dynamic_cast`. However, dwncasting from value to handle impossible even using `dynamic_cast` as the value object passed to `dynamic_cast` does not know of its handle object and may in fact have any number of handle objects referring to it.

6.9 Using declarations bringing in Value type members

With inheritance it is possible to bring otherwise hidden names into scope by a using declarations of the form:

```
using Base::value_name;          // Bring in values from base class, dependent or not.
using typename Base::type_name; // Bring in types from dependent base class.
using Base::Base;                // Bring in base class constructors.
```

Even disregarding dependent bases this is useful to avoid that subclass member functions hide base class member functions and to bring the overload sets together. For dependent bases it is a mandatory way to tell the compiler that the actual base will have these names as values and types respectively. Using declarations for base class constructors is often convenient when subclasses don't need to change their signatures.

It is proposed to allow bringing in names except constructor names into the handle type from the value type, bearing in mind that it is not recommended to declare member names in generic handle classes. Using declarations for constructors are however meaningless as the handle class does not construct the value class in a foreseeable way.

6.10 Copy/move assignment/construction

The existence of an ICF in a class does not affect how it is copy constructed or assigned. This offers full flexibility on how to handle these operations, and as this is not inheritance there is no base class construction or assignment to take care of.

6.11 sizeof, alignof

When applying `sizeof` and `alignof` on Handle types the ICFs do not affect the return value. As this is not inheritance there is nothing strange with the fact that the size of a Handle object can be less than the size of the return type of the ICF.

6.12 Protected member access

The fact that a class has an ICF returning objects of some class does not allow it to access `protected` members of the returned type. This is an arbitrary decision informed by the current use cases, which can be revisited if some rationale for allowing access to protected value class members can be found.

6.13 ICFs can be virtual

Non-implicit conversion functions may be virtual. This means that one can have a group of handle classes that return the same Value type and then take a reference or pointer to the handle class base class. Does not seem particularly useful but also it doesn't seem to increase complexity much, it's just that the dispatch to the ICF can be virtual, it doesn't change *when* it is called.

For the member pointer case it will add another complication though, where static or virtual dispatch to an ICF must be possible to select at runtime.

7 Examples

Here are a few examples of the usefulness of implicit conversion operators. Note that none of these are part of this proposal, which deals with the core language feature only.

7.1 The return type for string interpolation

A problem with std::format is that it returns std::string which is suboptimal if the intended operation is to stream the result of performing the formatting to for instance an ostream or into a pre-allocated buffer. Unfortunately we can't change the return type of std::format at this time but if we get string interpolation with f-strings there is

a new opportunity for optimization lost if we let it produce a `std::string`. To counter this the initial proposal for string interplolation (not yet published) added separate x-literals to serve this purpose. With a new type with an ICF returning `std::string` we don't need to have different literal prefixes to get optimal performance in all cases.

```
struct formatted_string {
    using auto = std::string;

    formatted_string(std::basic_format_string<char, Args...> fmt, Args&&... args) :
         m_fmt(fmt), m_args(std::make_format_args(std::forward<Args>(args)...)) {}

    implicit operator std::string() { return std::vformat(m_fmt.get(), m_args);

    std::basic_format_string<CharT, Args...> m_fmt;
    decltype(std::make_format_args(std::declval<Args>()...)) m_args;
};
```

Now, if an f-literal results in a `formatted_string` any use that requires a std::string the ICF is implicitly called, performing the formatting, while a new overload of `std::print` and similar functions can use the `m_fmt` and `m_args` members directly to optimize performance.

To avoid dangling references when the `m_args` member refers to temporary results of expressions inside the f-string the `using auto = std::string;` construct is used, but this requires the [N4035] proposal.

7.2 A universal lazy argument type

Any function that conditionally uses an argument with deduced type, such as `value_or` of the monadic API can benefit from a lazy evaluation wrapper in case the conditionally used value is costly to compute. Using an IFC a generic `lazy` wrapper can be designed so that the costly computation is performed on demand, without the called function (such as value_or) being aware of the lazy wrapper. This is not possible today.

```
template<typename F> struct lazy {
    lazy(F f) : func(std::move(f)) {}

    implicit operator decltype(auto)() { return func(); }

    F func;
};
```

7.3 A better reference wrapper

By making the user defined conversion function of std::reference_wrapper `implicit` the reference_wrapper can be used as the object it wraps in all cases, where today this is limited to when used as a function argument when the parameter type does not require further user specified conversions. There should be no major breaking changes caused by making this conversion function implicit as it only makes illegal code legal. Only the unavoidable SFINAE related breakages can happen (i.e. when SFINAE or constraints are used to detect that you *can't* do the things that this change enables). It is outside the scope of this proposal to determine if this is acceptable, but if it is it would increase the usability of reference_wrapper a lot, and if it isn't maybe a new parallel class like `copyable_reference` with this semantic should be added.

7.4 A more intuitive polymorphic/indirect

The new polymorphic/indirect class templates are prime use cases where it would be logical to use dot notation for access as they are not nullable except by moving from them.

A better approach could be to have a nullable copying_ptr and then complement that with a smart reference copying_ref as described below. This makes the set of "smart" classes orthogonal between copy semantics and

nullability.

7.5 Non-nullable smart references

It would be interesting to get `shared_ref` and `unique_ref` classes for the shared and unique semantics apart from the deep-copy semantics of polymorphic and indirect, but without the nullability of `shared_ptr` and `unique_ptr`. Likewise, a `pointer_ref` class would be a way to indicate that the pointer shall not be null, but still allow free copying. This is essentially the same as the better refererence_wrapper discussed above but with a name consistent with the other new classes described here.

While these types would have the same copy semantics as their smart pointer counterparts deep copying into the referred object is not as straight-forward as with pointers where you can easily dereference before assigning. Instead `static_cast<value_type&>(ref)` can be used, which will call the ICF. Alternatively `*&ref` can be used, employing the overloaded operator shown below.

Another thing that may be interesting is to get at the embedded pointer to for instance be able to compare pointers for equality rather than the referred objects, or to just continue using a `shared_ptr` without requiring `enable_shared_from_this` inheritance on the value_type. To enable this a `std::unwrap` friend function can be defined.

```
namespace std {

template<typename PTR> class universal_ref {
public:
    using value_type = pointer_traits<PTR>::element_type;

    universal_ref() = default;

    // Construct from the pointer-like, which must not be null.
    universal_ref(const PTR& src) pre (src) : m_ptr(src) {}
    universal_ref(PTR&& src) pre (src) : m_ptr(std::move(src)) {}

    // These conversions implement the operator.() functionality:
    implicit operator value_type&() & { return *m_ptr; }
    implicit operator const value_type&() const & { return *m_ptr; }
    implicit operator value_type() && { return std::move(*m_ptr); } // Maybe not: shared_ptr!

    // These are needed to hide universal_ref when operator& is used. Formulated as &*m_ptr to
    // let value_type::operator& kick in if it is implemented.
    decltype(auto) operator&() & { return &*m_ptr; }
    decltype(auto) operator&() const & { return &*m_ptr; }

    friend const PTR& unwrap(const universal_ref& src) { return src.m_ptr; }
    friend PTR unwrap(universal_ref&& src) { return std::move(src.m_ptr); }

private:
    PTR m_ptr;
};

// We could include type aliases for different smart and less smart pointers.
template<typename T> using shared_ref = universal_ref<shared_ptr<T>>;
template<typename T> using unique_ref = universal_ref<unique_ptr<T>>;
template<typename T> using pointer_ref = universal_ref<T*>;          // A more transparent reference_wrapp

}  // namespace std
```

— shared_ref and unique_ref requires `unwrap` to be used to access smart pointer methods such as `shared_ptr::use_count` or `unique_ptr::release` which is intuitive.
— shared_ref and unique_ref are created using make_shared and make_unique as they have constructors from the corresponding pointers.
— relational operators compare the values returned by the ICF as there is no comparision operators defined.
— The rvalue overload of the ICF returns by value in keeping with the str() && overload of stringstream. This avoids dangling references in case the returned value is captured by a reference for later use. This is somewhat problematic for `shared_ref` as we should only move from the pointee if `use_count() == 1`, but it is still logical as when you are moving a value from one handle to a shared value this moving is visible from all the handles.

Here are some usage examples.

```
shared_ref<MyClass> a = make_shared<MyClass>(1);
shared_ref<MyClass> b = make_shared<MyClass>(2);

a == b;                              // compare MyClass objects
std::unwrap(a) == std::unwrap(b);    // Compare pointers

a = b;                               // Assign between shared pointers

*&a = b;                             // Assign between MyClass objects. For b the ICF is called to convert.
static_cast<MyClass&>(a) = b;        // Assign between MyClass objects. For b the ICF is called to convert.

shared_ptr<MyClass> p;
shared_ref<MyClass> c = p;           // Assert, tried to construct from a null pointer!
```

As can be seen the assignment is not consistent with the comparison, but if the `operator=` was between objects the properties of sharing and cheap moving would be lost and require `std::unwrap(a) = std::unwrap(b)` which seems counter-intuitive given the class names.

8 Syntax alternatives

Some alternate syntaxes were considered before settling on the main proposal above. These alternatives were found to be more problematic than putting the intuitively understandable context-sensitive keyword `implicit` on a conversion function.

8.1 Explicitly stating inheritance by reference in the base class list

This syntax is a slight variation of the `class Sub : using public Base` syntax of [P0352R0] and has the same drawbacks.

The drawback with the proposed syntax in this proposal is that it doesn't show clearly that we should think of inheritance, as the value type is not in the base class list. A variation of the syntax is to state that the handle inherits by reference. Unfortunately this is not enough, we still need to write the code that returns the T& somewhere. This would reasonably be in a regular conversion function:

```
template<typename T> class Proxy : public T& {
    Proxy(T& object) : m_ptr(&object) {}

    operator T&() { return *m_ptr; }
    operator const T&() const { return *m_ptr; }

private:
    T* m_ptr;
};
```

The drawback of this is that we have to write two things in concert at different places in the class definition. On the other hand cast operators is already a well known concept.

The advantage of this formulation is that there is no extra keyword and that we can specify private inheritance if we want, although it is hard to see a use of this as then the outside user would not be able to access the value anyway.

A drawback is that it would be a harder sell to allow T to be int if T is mentioned in the *base class* list. Similarly the non-similarities mentioned above that you can't override methods but can create ICFs returning final classes speak against putting the T in the base class list, as does the fact that it blocks the possibility of returning an instance of a nested class from the ICF.

There are more exotic possibilities like putting the body of the cast operator in the base class list:

```
template<typename T> class Proxy : public T{ return *m_ptr } {
    Proxy(T& object) : m_ptr(&object) {}

private:
    T* m_ptr;
};
```

While this is terse it seems rather odd to have code in the base class list, and it is certainly much bigger change in the compiler.

8.2 Just allowing inheritance from references

While it may seem tempting to let the T& in the base class list just result in a T& subobject being created automatically, with the current rule that references must be initialized (in this case in the constructor). However this severely limits the usability of the feature as it does not allow for replacing the T object, it always refers to the same object it was initialized to.

Extending this to ability to inherit from anything, for instance `unique_ptr<T>` is not feasible as this is already possible with another meaning. Inheriting from `unique_ptr<T>&` would be possible but then there must be some rule to prevent this from resulting in a base class subobject of the reference type. This could be feasible and then the Handle class would inherit T*& which would result in a base class subobject of type `T*` but then the language must automatically dereference the subobject to find the names supposed to be made visible by the indirect inheritance, which makes the feature obscure and less general.

Another try would be to inherit with a leading * which is to be interpreted as the following type being the type of a base class subobject but where operator* is always applied to it automatically. However, to manipulate the *pointer-like* itself requires some other syntax which is not easy to define.

8.3 Circling back to operator.()

The importance of this proposal is not the spelling of the feature, but that its name lookup semantics is defined in terms of inheritance. There is actually nothing to prevent the spelling from being `operator.()` as proposed long ago. This has the advantage of avoiding a new keyword and being intuitive up to a point. The fact that with this semantics applying a dot does not invoke `operator.()` if there is a matching name in the Handle is however less intuitive. Also, while `f(a)` is natural if it calls a conversion function to convert `a` from handle to value calling `operator.()` to accomplish a type conversion is very strange.

9 Implementation experience

None so far.

10 Wording

There is no wording yet, but as this proposal claims to be easier to word than an operator.() proposal here is a sketch of how the wording changes could be introduced:

— Define the term ICF and the grammar addition needed to be able to declare one.

— Describe the rules for returning derived and base classes from ICFs.

— Define a term for base classes and ICF return types together.

— Change the use of base class to this new term in sections about name lookup, overload resolution and the scope resolution operator.

— Do not use the new term in the section about implicit upcasting of pointers but add a parallel section detailing that upcasting of pointers with ICFs requires the returned type to be a reference type.

— Change the use of base class to this new term (where possible) or add clauses for ICFs in the sections about member pointers. Do not change the leeway given to implementations to not allow arbitrarily complex member pointer types, but maybe recommend that compilers at least allow one ICF call *or* one virtual base indirection in addition to multiple-inheritance related offset additions.

— Augment the section on using declarations to use the new term except when discussing using declarations of base class constructors.

— It may be needed to mention that name lookup is ambiguous if handle and value inherit the same base class *virtually*, but maybe this is somehow covered by existing wording.

11 References

[N4035] P. Gottschling, J. Falcou, H. Sutter. 2014-05-23. Implicit Evaluation of "auto" Variables and Arguments.
https://wg21.link/n4035

[N4477] Bjarne Stroustrup, Gabriel Dos Reis. 2015-04-09. Operator Dot (R2).
https://wg21.link/n4477

[P0352R0] Hubert Tong, Faisal Vali. 2016-05-30. Smart References through Delegation: An Alternative to N4477's Operator Dot.
https://wg21.link/p0352r0

[P0416R1] Bjarne Stroustrup, Gabriel Dos Reis. 2016-10-16. Operator Dot (R3).
https://wg21.link/p0416r1

[P0700R0] Bjarne Stroustrup. 2017-02-21. Alternatives to operator dot.
https://wg21.link/p0700r0