WG21-P0416
2016-9-10

Follow-up to WG21-N4477
2014-10-11

Operator Dot (R3)

Bjarne Stroustrup (bs@ms.com)

Gabriel Dos Reis (gdr@microsoft.com)

What's new?

This is an update on N4477 primarily based on CWG feedback:

- Clarifies the meaning of smart references (i.e., a class with an **operator.()** defined): "A smart reference automatically dereferences by calling an **operator.()** when an operation that is not defined for it is applied to it."
- Confirms that smart references can be passed by copying, rather than always decaying to the referred-to type.
- Confirms that an initializer for an **auto** object and a template argument are deduced to the smart reference itself, rather than to the referred-to type.
- Clarifies overloading issues.
- Clarifies what happens when a class declares both an **operator.()** and a conversion operator.
- Clarifies the effects of inheritance: there are no new lookup rules.
- Clarifies that (unlike built-in references) smart references are objects. In particular, we can have arrays of smart references and **sizeof(Ref)** is the size of the **Ref**.
- Changes the meaning of smart reference assignment: by default assignment copies the handle rather than the value. To get assignment to copy the referred-to value (like built-in references do), you need to define the copy assignment.
- A smart reference follows the usual rules for generating default operations.
- We eliminated discussions related to earlier versions of the design.

Remember, this is a design document listing the major design decisions and giving rationale. The Wording for the earlier variant of the proposal is found in P0252R1; this needs to be updated based on CWG feedback.

Abstract

This is a proposal to allow user-defined operator dot (**operator.()**) so that we can provide "smart references" similar to the way we provide "smart pointers." The gist of the proposal is that if an **operator.()** is defined for a class **Ref** then **Ref** is a smart reference and every operation not declared for **Ref** is forwarded to the result of **operator.()**. Roughly, **operator.()** is to a smart reference what dereferencing is to a built-in reference. Note that by default a class has a copy constructor so pass-by-smart-reference is defined.

1 Introduction

There has been several suggestions and proposals for allowing programmers to define **operator.()** so as to be able to provide "smart references" similar to the way we provide "smart pointers" (e.g., [Adcock,1990], [Koenig&Stroustrup,1991], [Stroustrup,1994], and [Powell,2004]). Consider how that idea might work:

```
template<class X>
class Ref {
public:
        Ref(X& x) :p{&x} {}  // refer to x
        X& operator.() { /* maybe some code here */ return *p; }
        void rebind(X& x) { p=&pp; }  // refer to x
        // …
private:
        X* p;
};

X xvar {77};
Ref<X> r {xvar};        // make r refer to xvar

r.f();                  // means (r.operator.()).f() means (*r.p).f()
++r;                    // means ++r.operator.()
Ref<X> r2 = r;          // r2 refers to the same X as r (copy constructed)
```

Now **Ref<X>** is a proxy for an **X** object: The **Ref<X>** behaves like an **X** object, yet does not expose pointer-like behavior to its users. This **Ref<X>** does not manage the lifetime of "its" **X** (see also §4.7).

However, is that **++r** right? There is no mention of dot. Is it right to apply operator dot when the dot is not explicitly mentioned? This has been one sticking point for earlier proposals. On the one hand, we want to apply operator dot for "all uses" so that we can get operators, such as **=**, **+**, and **++** to work for "smart references" to objects of classes with such operators (like for built-in references). On the other hand, we also want to be able to operate on the state of the smart reference itself (a smart reference really is an object). For example:

```
X xvar2 {99};
r.rebind(xvar2);        // means r.p=&xvar2
```

This proposal is to apply operator dot everywhere, except when we “say otherwise.” So how do we specify an exception to the forwarding to the referred-to object? Many alternatives have been discussed in the various proposals, including:

1. Separate operators (e.g., **:-** for assignment to the reference object, like Simula)
2. Explicit use of **->**
3. Preference to members of the smart reference over members of the referred-to object
4. Define operator **.***
5. Use inheritance
6. Use template metaprogramming
7. Use overload resolution across the reference/referred-to scopes
8. Mark handle member functions as forwarded or not

None of the suggestions are perfect and we will not repeat those discussions. If every operation on a reference is forwarded to the referred-to object (as for a built-in reference), no trickery within the current language will give us a perfect solution: For example, we could not implement **rebind()**. On the other hand, if an operation is not forwarded, then we can’t invoke that operation on the referred-to object. For example, we cannot define **rebind()** on the smart reference and also have a call to **rebind()** automatically forwarded to the referred-to object. Something has to give.

2 Why do we want to “overload dot”?

Part of the problem of designing an **operator.()** mechanism is that different people have different views of what problem it is supposed to solve. Another is that since overloading of dot doesn’t exist, people imagine it might be tweaked to solve an amazing variety of problems. Here is a list of some suggested problems/solutions (not necessarily compatible, orthogonal, general, reasonable, or well-specified):

1. *Smart references*: This is the most commonly mentioned need. That is, a class that acts like a reference, but provides some extra service, such as rebinding, loading from persistent storage, or pre- and post-actions. In particular, = should apply to the referred-to object, not the handle.
2. *Smart pointer work-alikes*: That is, something that acts like an overloaded ->, but doesn’t have pointer semantics. So **.** (dot) should work like **->** does for smart pointers. In particular, **=** should apply to the handle.
3. *Proxies*: That is, something that acts just as an object (like a reference), but requires computation on access. A proxy is not necessarily a handle to some other object.
4. *Interface refinement*: That is, provide an interface that adds and/or subtracts operations from the interface provided by a class. Such an interface is not necessarily a handle to some other object.
5. *Pimpl*: That is, providing access to an object through an interface that provides a stable ABI. For example, changing the objects layout doesn’t affect a user.
6. *Handles*: That is, anything that provides indirect access to something else.

Sometimes, an idea is described in very general terms and sometimes just as a single use case. Some examples are shown below (e.g., §5).

So **operator.()** refers to a language mechanism supposed to help writing classes. One common theme is "but what I have (want to have) is ***not*** conceptually a pointer so I don't want to use the **->** or ***** notation to access values." What is also common is the idea to have an operation applied to a "handle" actually be applied to a "value object" without actually listing every possible operation from the value's type in the definition of the handle's type. Not all such handle/value ideas can be supported by a single **operator.()** mechanism. From now on, we will use "handle" to refer to a type with **operator.()** defined and "value" to the type that **operator.()** forwards to (and to objects of those types).

On the other hand, a smart reference (the handle) is an object (unlike a built-in reference), so that we can have an array of smart references and so that a smart reference can have a modifiable state (e.g., for rebinding). A smart reference is typically not just a conversion to the value to which it refers.

2.1 Why now? (again)

With that many proposals and suggestions, why raise the issue again? After all, if it was easy to come up with a widely acceptable design it would have been accepted long ago. When something that is frequently requested, is frequently commented on as a flaw in the C++ design, and is widely considered fundamentally a good idea fails, we should try to learn from the failures and try again. Furthermore, the importance of non-indirection (proxies) has increased over the years, as has the importance of limiting raw pointer use. **Operator.()** is the last piece of the puzzle of how to control the lifetime and use of objects without relying on application users being well-behaved. Also, this design includes a couple of new ideas.

3 An operator.() design

We conjecture that the key to an **operator.()** design is just six operations:

- **operator.()** – defines the meaning of forwarding. We need to decide whether it applies only to explicit uses of dot (.) or also to implicit ones. This design forwards in both cases.
- **&** – does it apply to the handle or the value? Does a programmer have a choice? Can it be used to gain access to the address of the value object? Some consider that most undesirable. Can it be used to return a "smart pointer" to the object referenced by a "smart reference?"
- **=** – does it assign handles or values? Does a programmer have a choice? This design gives the programmer a choice. By default, **=** applies to the handle, but we can define the handle's assignment to apply to the value.
- **rebind()** – does a named function apply to the handle or the value? This design makes a handle member functions apply to the handle and all others apply to the value.
- Initialization – do initialization of a handle use the handle's constructors (in particular, the copy constructor) even if they are not explicitly declared? It does, so we can pass-by-smart-reference similar to the way we pass-by-reference (the default copy constructor copies).

- Type deduction – does a smart reference deduce to the reference itself or the value to which it refers when passed as a template argument or used to initialize and **auto**? It deduces to itself (the handle) since there is no operation to forward to.

So, the **Ref<T>** example works exactly as written.

4 Design points

This section considers the proposed design and a few alternatives.

4.1 Implicit dot

Why do we want forwarding (application of **operator.()**) to expressions that do not use dot? Consider:

```
void f(X& x, X& y)
{
        If (x!=y)
                x=++y;
}
```

For ordinary references **x** and **y**, the comparison, assignment, and increment applies to the referred-to objects. The claim is that we want the same for

```
void f(Ref<X> x, Ref<X> y)
{
        If (x!=y)
                x=++y;
}
```

For every **Ref<X>** that we can think of where **X** has **!=**, **=**, and **++,** etc., we do not want to have to write:

```
void f(Ref<X> x, Ref<X> y)        // explicit forwarding
{
        if (x.operator.()!=y.operator.())          // ugly!
                x.operator.()= ++y.operator.();
}
```

However, we want the original code (above) to mean that.

Why do we ever want to apply operations to the "smart reference" itself? That is, why don't we just forward ***every*** operation? One reason for wanting smart references is to be able to have them more flexible than built-in references. The **rebind()** operation is a common example.

Operator dot is not invoked for explicit uses on the **->** operator on a pointer. For example: the compiler will not rewrite **p->x** to **(*p).x** and then to **(*p).operator.().x** if ***p** is of a type that has a defined **operator.()**. This rule is needed to allow us to define **operator.()** and **operator->()** separately and to allow us to use pointers in the definition of an **operator.()**. This is the same rule as for other user-defined operators. For example defining **+** and **=** does not implicitly give us **+=**. It is up to the programmer to define consistent sets of operators. It is also the rule we use for **operator->()**: when **b** is a smart pointer, **p->m** is not rewritten to **(*p).m** so that **operator** is never called.

4.2 Pass by smart reference and type deduction

By default a class has a copy constructor. Consider:

```
void g(Ref<X> r)
{
        X x1 = r;          // X x1 = r.operator.();
        X& rx = r;         // X& rx = r.operator.();
        Ref<X> r2 = r;     // rx1 refers to the same X as x (use Ref<X>'s default copy constructor)
        // …
}

X xvar { 99 };
g(xvar);                   // construct a Ref<X> from xvar and pass it to g()

Ref<x> rr {xvar};          // construct a Ref<X> from xvar
g(rr);                     // call (rr); pass-by-smart-reference
```

Consider further:

```
template<typename T>
void fct(T x1, T& x2, Ref<T> x3)
{
        T xx1 = x1;        // T xx1 = whatever x is
        T xx2 = x2;        // T xx2 = whatever x2 refers to
        T xx3 = x3;        // T xx3 = whatever x3 refers to: x3.operator.()
}

X x {9};
X& r = x;
Ref<X> rr = x;

fct(x,x,x);        // fct<X>: pass-by-value, pass-by-reference, pass-by-smart-reference
fct(r,r,r);        // fct<X>
fct(rr,rr,rr);     // fct<X>
```

In each case, the references are dereferenced and the **X** arguments are passed by-value, by-reference, and by-smart-reference.

Note that we deduce **T** in **Ref<T>** to be **X** from an argument of type **X**. That's a C++17 feature from P0091R1.

But what does it mean to pass to a smart reference? That depends on what is the meaning of initializing a **Ref<X>** with an **X**. The designer of **Ref** has a choice to make:

- Assume that **Ref<X>** doesn't own the object (like **X&**), so that **Ref<X>**'s destructor does not delete the **X**. That is what **Ref<X>** as defined above does.
- Assume that **Ref<X>** owns its **X** (and must delete it). Most likely, **Ref<X>**'s copy constructor must clone an **X** argument, but that is quite different from what an **X&** does for initialization (including argument passing). That semantics is usually associated with things called something like **Proxy<X>**.
- Assume shared ownership and use a use count (manipulated in **Ref**'s copy constructor).
- Don't support construction of an **Ref<X>** from an **X**.

This design choice is quite similar to what we have to make for smart pointers (no ownership, unique ownership, or shared ownership). This is to be expected.

4.3 Type deduction

When we pass a smart reference as a template argument or initialize an **auto** object, we must decide which type to deduce to: the handle or the value? We deduce to the handle. For example:

```
template<typename T>
void fct(T x1, T& x2, Ref<T> x3)
{
        T xx1 = x1;       // T xx1 = whatever x is
        auto r1 = x1;     // r1 is a T

        T xx2 = x2;       // T xx2 = whatever x2 refers to
        auto r2 = x2;     // r2 is a T

        T xx3= x3;        // T xx3 = whatever x3 refers to: x3.operator.()
        auto r3 = x3;     // r3 is a Ref<T> as for any other clas
}

X x {9};
X& r = x;
Ref<X> rr = x;

fct(x,x,x);
fct(r,r,r);       // fct<X>
fct(rr,rr,rr);    // fct<X>
```

Why should a smart reference differ from a built-in reference for **auto** deduction? It does so because the use of **operator.()** is triggered by a use that must be forwarded, but there is no use until after the deduction. This is a design point for which argument can be made either way; see §5.6 for an example and further discussion.

4.4 Who is in control?

The control of whether a function **f()** is applied to the handle or to the value can be vested in the handle or left to the user. This design vests the control in the handle. A handle without an operation could make sense, but its behavior would be fixed at constructions time. However, like **Ref**, most examples need operations on the handle. The question is then how to define and apply them.

This design took what seems the simplest approach: an operation defined as a member of the handle is an operation on the handle. Otherwise, the operation is forwarded. This also seems to lead to the simplest and “most natural” code.

Note that every name “reserved” by declaring it in the handle, takes away that name for some value type. In particular, by declaring **rebind()** for a handle **Ref<T>**, we cannot access that function in the **Ref<X>** of a **Ref<Ref<X>>**.

We considered the alternative of always forwarding except when a function was applied through a pointer. However, unless more rules are invented, **&** is also forwarded to the value, so how do we get a pointer to the handle? Also, having **x.f()** potentially mean something different in different contexts would be quite confusing to users. We propose to use the access-through-pointer trick in the implementation of handles, though.

4.5 Why give priority?

Why give priority to members of the handle? We could

1. Give priority to members of the handle
2. Give priority to members of the value
3. Give an error if a name is declared in both the handle and the value
4. Apply overload resolution to choose between members of the handle and the value
5. Provide a syntax for access to the handle

We consider option 1 (“handle priority”) the right choice in most cases, and the simplest solution. Choosing this option implies that nothing can be done to access the handle unless an operation has been declared (explicitly or by default) in the handle type (except through a pointer). This way, the writer of the handle (and its users) can easily know which operations are offered by the handle – there is no need to examine the value to determine the set of operators on the handle.

Option 2 (“value priority”) simply does not make sense because the purpose of the rule is to allow handle members to be invoked; value member access is the default.

Option 3 (“either or”) would be brittle and confusing. Also, in general a class author or a class user would not know if a construct was correct until template instantiation time because the value type will often be a template argument.

Option 4 (“overload resolution”) is technically not too bad: The compiler simply considers the union of overload sets from two scopes. However, it can be hard for the programmer to know which overload is chosen (the details of the value and handle types may not be known to a user and are in separate in the code), it opens the field for all sorts of cleverness, and suffers from the problem of potentially delaying answers until template instantiation time. Also, for a templatized handle, the operations available on the handle could easily differ from instantiation to instantiation depending on the operations provided by the template arguments.

The choice between (1) and (4) is fundamental from a language-technical perspective, but unlikely to be important in real code.

Option 5 ("access syntax") would allow us to define operations on the handle without hiding operations on the value. But how? Consider what we might do if we did not give priority to members of the handle:

```
void f(Ref<X> r, X& x)
{
        adderssof(r)->bind(x);  // go through pointer (one alternative)

        Ref<X>::r.bind(x);      // explicit qualification (another alternative)

        r..bind(x);             // .. handle access operator (a third alternative)
        r .= r2                 // .= might make bind() redundant
}
```

We could

1. *go through a pointer*. Like all operations, **&** is forwarded to the value through **operator.()**, so getting into the handle through a pointer requires **std::addressof()**, which exists specifically to defeat smart pointers.
2. *use explicit qualification* (with **::**)to defeat forwarding through **operator.()**, just as we use it to defeat virtual calls, but that's rather verbose and ugly.
3. *introduce some new syntax* to distinguish operations on the handle from operations on the value. Here I have used a suffix dot followed by whatever would normally have been written.

The "**addressof()** trick" will work whatever else we choose to do.

Note that the priority is simply "has a member of that name been declared in the handle?" Like for access control, we don't distinguish based on the type of the member (e.g., function vs. data member) or try to do overload resolution between scopes. Thus an "obviously irrelevant" member of the handle (e.g., a private member or a type name) can hide a member of the value. This problem would also affect the overload resolution solution (4). This problem could be solved by a change in the lookup rules. The same problem surfaces in the context of modules, and will almost certainly have to be addressed by any module proposal. Our assumption is that the problem is manageable and is likely to be eliminated in the context of a module proposal. So here, we don't propose a solution.

4.6 Inheritance

"Has been declared in the handle" includes names found in the handle through Inheritance. Consider:

```
struct Base {
        X operator.();
};

struct Derived : Base {
        // …
};
```

Here, **Derived** has an **operator.()**.

Similarly, there are no new lookup rules when determining if a smart reference has a function declared. For example:

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

struct Derived : Base {
        X operator.();
        // …
};

Derived d = …;
d.f();    // call Derived's f(); that is Base::f()
```

Here, **Derived** "has an **f()**" and we would not call an **f()** declared in **X**.

4.7 Smart references are objects

A built-in reference is not an object. That is, when you take its address, you get a pointer to the referred-to object, when you take its size, you get the size of the referred to object, and you cannot have an array of references. In contrast a smart reference is an object by virtue of being a class type:

```
Ref<X> r = x;
auto s = sizeof(r);             // the size of the handle
assert(s == sizeof(Ref<X>));    // OK
Ref<X> a[10];                   // OK
auto addr = &x;                 // by default a X*, but you can overload & for the handle
```

People have expressed a desire for having arrays of smart references, and that seems to be the right choice. If we (like for built-in references) decided to have **sizeof(r)** return the size of the value, this would not be possible.

4.8 Constructors and destructors

A constructor invocation does not forward (invoke **operator.()**) because there isn't an object to forward from/through until the constructor completes.

A destructor invocation does not forward (invoke **operator.()**) because a destructor reverses the action of a constructor and the constructor does not forward.

4.8.1 Unsurprising consequences

To explore implications, consider again **Ref<X>** as defined in §1.

```
template<class X>
class Ref {
public:
        Ref(X& x) :p{&x} {}  // refer to x
        X& operator.() { /* maybe some code here */ return *p; }
        void rebind(X& x) { p=&pp; }  // refer to x
        // …
private:
        X* p;
};
```

We can try

```
void f(Ref<X> rr);

X x;
Ref<X> r {x};

f(x);   // OK: use constructor (Remember P0091R1)
f(r);   // OK: use Ref's copy constructor
```

Now consider defining a **Ref2** that owns its value object, much as **unique_ptr** owns its object:

```
template<class X>
class Ref2 {
public:
        Ref2(X& x) :p{new X{x}} { }  // make a copy of x
        X& operator.() { /* maybe some code here */ return *p; }
        void rebind(X& x) { auto np = new X{x}; delete p; p=np; }  // make a copy of x
        ~Ref2() { delete p; }
        // …
private:
        X* p;
};

void f(Ref2<X> rr);

Ref2<X> r {x};
X x;

f(x);   // OK: use constructor (Remember P0091R1)
f(r);   // OK: no copy constructor, but r.operator.() is an X so f(X{r.operator.()})
```

Declaring a destructor suppresses the copy constructor so **Ref2** forwarded using **operator.()**. An application of the default copy constructor in this case would have led to a memory leak. Since the use of a copy constructor here is only deprecated, rather than banned, we have better be explicit:

```
template<class X>
class Ref2 {      // version 2
public:
        Ref2(X& x) :p{new X{x}} { }       // make a copy of x
        Ref2(const Ref&) =delete;         // no copy construction
        X& operator.() { /* maybe some code here */ return *p; }
        void rebind(X& x) { auto np = new X{x}; delete p; p=np; }  // make a copy of x
        ~Ref2() { delete p; }
```

```
		// …
	private:
		X* p;
	};
```

Had **Ref2** not had a constructor that accepted an object returned by its **operator.()**, **f(r)** would have failed.

Alternatively, we might have defined a copy constructor:

```
	template<class X>
	class Ref2 {	// version 3
	public:
		Ref2(X& x) :p{new X{x}} { }	// make a copy of x
		Ref2(const Ref& r) { auto np = new X{r}; delete p; p=np; }  // copy r.operator.()
		X& operator.() { /* maybe some code here */ return *p; }
		void rebind(X& x) { auto np = new X{x}; delete p; p=np; }  // make a copy of x
		~Ref2() { delete p; }
		// …
	private:
		X* p;
	};
```

An alternative implementation would make **p** a **unique_ptr**.

Similarly, we could define a **Shared_ref** (use counted shared object):

```
	template<class X>
	class Shared_ref {
	public:
		Shared_ref(const X& r) :p{new X{r}} {}
		X& operator.() { /* maybe some code here */ return *p; }
		// …
	private:
		shared_ptr<X> p;
	};
```

It will probably be wise to follow the rule that if a copy operation, a move operation, or a destructor is explicitly declared, all should be. We considered having the rule that if **operator.()** is declared, no special functions are declared, but that seemed too Draconian.

Note that to get assignment to apply to the value, rather than the handle, we need to define operator=() on the handle to do that (see §5.6).

4.9 Recursive use of operator.()

Is **Ref<X>::operator.()** applied to uses of a **Ref<X>** within members of class **Ref<X>**? Consider a **Ref** that owns it **X** so that it must implement copy and move operations:

```
template<class X>
class Ref {
public:
        Ref(const X& x) :p{new X{x}} {}
        X& operator.() { /* … */ return *p; }
        Ref(const Ref& a);        // copy constructor: clone *a.p
        Ref(Ref&& a)              // move constructor: replace p with a.p
                :p{a.p} { a.p=nullptr; } // error: p= a.operator.().p ? that is, p=a.p.p
        // …
private:
        X* p;
};
```

This design has no special rules for handle members, so the answer is “yes, **operator.()** will be called,” so the definition of the move constructor is an error: **a.p** is interpreted as **a.operator.().p** which means **a.p.p** which is a compile-time error. The reasons for this design decision are that

- a context-dependent rule could be surprising and difficult to implement
- a context-dependent rule would be as surprising to some as the lack of one would be for others
- it is unlikely that **Ref<X>** objects will be common in **Ref<X>** definitions (the copy and move operations are likely to be the most common cases)
- like for smart pointers, the implementers of “smart reference” classes are likely to be relatively few and relatively expert compared to the enormous number of users
- errors from forgetting to use pointers to access the value are most often caught by the compiler
- “no rule” is the simplest rule

This (lack of a rule) implies the need to use pointers in many handle implementations. For example:

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

Even if the handle defined **operator&()**, say to return a “smart pointer” and/or to prevent the pointer to the value to leak into the surrounding program, we must use **std::addressof()**. For example:

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

The standard library **addressof()** is guaranteed to return a built-in pointer rather than a smart pointer. Note that the compiler will not implicitly transform **p->x** into **(*p).x** and allow a further transformation to **(*p).operator.().x** (§4.1). That way, we do not get into a recursive mess. When naming a member (such as **Ref<T>::p**) from within the class (as in ***p** in **operator.()**) the name is interpreted as prefixed by **this->** (here, ***(this->p)**). To avoid a recursive mess, we must not reduce that **this->p** to **(*this).p** and further to **(*this).operator.().p**.

If we need a pointer repeatedly, we need to use **addressof** only once:

```
X* q = addressof(a)->p;
```

4.10 Return type of operator.()

The return type of **operator->()** is required to be something to which **->** can be applied. This restriction is not fundamental. We propose not to impose the equivalent rule for **operator.()** (and propose also to relax it for **operator->()**): Allow **operator.()** to return a value of a type for which dot is not defined, such as an **int**. Consider:

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

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

This can come in handy. However, consider the proposals to allow **x.f()** to invoke **f(x)** (Glassborow,2007], [Sutter,2014], [Stroustrup,2014], [Stroustrup&Sutter,2015]), generalizing what we already do for operators, such as **==**, and for **begin()** in a range-**for**. Should such a proposal ever succeed, we would be able to invoke non-member functions through **operator.()**: If **x.operator.().f()** does not correctly resolve to a member function **X::f()**, we try to resolve it to **f(x.operator.())**. For example:

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

S s {7};
int x = s.sqrt();  // s.operator.().sqrt() resolves to  sqrt(s.operator.()) that is sqrt(s.a)
```

Since **int** has no member functions, **s.operator.().sqrt()** makes no sense, so we try **sqrt(s.operator.())** and succeed.

As for **operator->()**, errors relating to the return type should be diagnosed only if the **operator.()** is actually used.

4.11 The definition of operator.()

There are no specific restrictions on to definition of **operator.()**. For example, we might implement a version of the "wrap-around" pattern [Stroustrup,2000]:

```
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 Wrap(p); }
        // …
private:
        X* p;
};

void foo(Ref<X>& x )
{
        x.foo();                // x.operator.().foo()
                                // Wrap(x.p).foo()
                                // before(); acess(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()
}
```

The usual scope rules ensure that the two **operator.()**s won't get confused (by the compiler).

4.12 Overloading operator.()

Since **operator.()** doesn't take an argument, overloading seems implausible. However, to cope with **const**, we must at least be able to overload on **this**. For example:

```
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()
}
```

This is simply the usual rules applied, as for **operator->()**.

Beyond that, we can allow selection based on how a set of **operator.()** were used. Consider:

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

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


struct S3 {
        T1& operator.() { return p; }  // use if the name after . is a member of T1
        T2& operator.() { return q; }  // use if the name after . is a member of T2
        // …
private:
        T1& p;
        T2& q;
};

void (S3& a)
{
        a.g();              // error: ambiguous
        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")

        auto x0 = a.m;            // error: ambiguous
        auto x1 = a.m1;           // a.p.m1
        auto x2 = a.m2;            // a.q.m2
    }
```

Here, the compiler looks into **T1** and **T2** (the return types of the two **operator.()**s, select the most appropriate member from either (if any) and then use the appropriate **operator.()** for that member's class. Member selection is done by ordinary overload resolution between the results of the two **operator.()**s. Note that we look into **T1** and **T2** individually, rather than into the union of the scopes of **T1** and **T2**. This (among other things) allows us to have a simple proxy to an "object" composed of several separately allocated parts.

Resolving an expression involving a smart reference can involve several operations:

1. When we look for an **operator.()**, we may find it in a base class. That's a built-in conversion.
2. When we have found an **operator.()**, it may return a type with **operator.()** defined, so we can find a sequence of **operator.()**s. The application is a bit like a conversion (and like a dereference of a built-in reference) but we don't count it as a conversion.
3. When we find the final value, a conversion may be needed to invoke the operation.

We don't propose to impose any new restrictions on the number of conversions. Following a chain of **operator.()**s is a recursive application of the rules for resolving a single use an **operator.()**.

4.14 Conversions

If we define both an operator.() returning a T and a conversion operator to T, we can have an apparent ambiguity:

```
    struct Ref {
        Ref(X&);
        X operator.();
        operator X();
        // …
    };

    X x;
    Ref r {x};
    X x {r};  // operator.() or operator X()?
```

Because **operator X()** is an operation declared on the handle, it takes priority over **operator.()**:

```
    X x {r};  // operator X()!
```

If both operations are defined for a type they are likely to be identical.

4.15 Member Types

In addition to data and function members, a class can have type members. Can these be accessed through **operator.()**? No. This proposal only applies to names that could be accessed using dot (**.**). Consider:

```
    struct S {
            using T = int;
            // …
    };

    S s;
    s.T x;    // error: we can't access a member type using dot
```

Adding an **operator.()** to **S** would make no different to that example.

As usual, we can use types in combination with **::** for disambiguation. For example:

```
struct B1 { int a, b1; };

struct B2 { int a, b2; };

struct S : B1, B2 { };

void ff(S& s)           // traditional use
{
        s.b1 = 7;
        s.b2 = 8;
        s.a = 9;        // error: ambiguous
        s.B1::a = 10;
        s.B2::a = 11;
}
```

The rules for using **S** through a "smart reference" are unchanged from those for using it through a built-in reference:

```
void ff(Ref<S> s)       // use through "smart reference"
{
        s.b1 = 7;
        s.b2 = 8;
        s.a = 9;        // error: ambiguous
        s.B1::a = 10;   // s.operator.().B1::a = 10
        s.B2::a = 11;   // s.operator.().B2::a = 11
}
```

This technique achieves for composition using static calls what multiple inheritance archives with virtual calls. Smart references have less complexity and lower run-time overhead.

56 Examples

Here are a few examples of possible uses of **operator.()**.

5.1 Pimpl

This pattern separates a stable interface from a potentially changing implementation in a way that does not require recompilation when the implementation changes. There are many variants of Pimpl. For example:

Here is the supposedly stable part:

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

Only the public interface (here **int foo();**) is used through the handle:

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

Here is the potentially less stable implementation:

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

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

A use where all access to an **X** goes through **Handle<X>**'s pointer:

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

int main()
{
        Handle<X> hx{ new X{ 7 } };
        f(hx);
}
```

The members of **X** need not be virtual, but could be. Virtual functions, or other trickery, could be used to eliminate implementation details from the header seen by **operator.()**.

5.2 Adding operations in a proxy

Consider how we might provide a standard interface for a class. We can provide a set of functions as an interface, adding it to whatever else a class might offer:

```
template<typename T>          // T must have a <
struct totally_ordered {
        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); }

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

struct basic_ordinal {
        BasicOrdinal(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>;
```

It is a good guess that a simple inline **operator.()** , like the one above, will be inlined. Thus, the use of **totally_ordered** incurs no overhead compared to handcrafted code.

Naturally, this could be done better with concepts, and be expressed far simpler with the proposed terse syntax for comparisons [Stroustrup,2014]. However, this is an example of a general technique, rather than a recommendation of how to do operator functions.

5.3 A remote object proxy

Here is a simple remote object proxy that reads a copy of a symbolically-named remote object into main memory upon construction and back again upon destruction:

```
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;
};
```

5.4 Optional

Here is a simplified **Optional** implementation (we capitalize to emphasize that we are not aiming for compatibility with **std::optional**):

```
template<typename T>
class Optional {
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 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 initialization of obj
        };
        bool b;
};
```

Obviously, a lot of details are missing, but what we are interested in here is the use and interaction between the handle and the value.

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

complex<double>>& r0 = oz0;   // throws: oz0.operator.() finds oz0.b==false
complex<double>>&  x1 = oz2;  // x1 is {1,2}

auto z0 = oz0;                // z0 is an Optional
```

```
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}; })
```

Using **Optional<X>** exactly as an **X** is possible, but would probably lead to too many **obj_empty()** tests, therefore we provide **obj_empty()** as a public function. This is an example of where the "handle priority" policy matters. We chose the slightly awkward name **obj_empty**, rather than a popular name, like **valid**, to make name clashes between the handle and the value type less likely.

It would have been nice if we could have overloaded **operator.()** on lvalue vs. rvalue. Then, we could have eliminated the **operator=**. Note that we are relying on **operator.()** being used on all accessed not mashed by the handle, even the simple read of an **Optional<X>**.

5.6 An ordinary reference

Can a smart reference be defined to do what a built-in reference can do? Almost; there are a few "magic" properties of a built-in reference that cannot be matched exactly. However, seeing how close we can get is a good test of the mechanism, just as trying to define a class **Int** that behaves like a built-in **int**.

```
template<class X>
class Ref {
public:
        Ref(X& r) :p{&r} {}
        X& operator.() { return *p; }
        X& operator=(Ref& arg) { return *p = *arg.p; }  // assign values!
        Ref(const Ref& arg) :p{arg.p} {}                // assign handles
        // …
private:
        X* p;
};

X x {111};
Ref<X> r {x};      // smart reference to x
X& rx {x};         // built-in reference to x

auto r2 = r;       // r2 is a copy of x
auto rx2 = rx;     // rx2 refers to x (!! Note)

X& rx3 = rx;       // rx3 also refers to x
Ref<X> r3 = r;     // r3 also refers to x

R2 = x;            // copy values: *r2.p = x
```

```
R2 = x;        // copy values: r2.operator=(Ref<X>{x}); that is *r2.p = *Ref<X>{x}.p
```

Binding to a **Ref** doesn't extend the lifetime of an **X** the way binding to an **X&** does.

We decided against having **auto** deduce the value type (as the built-in reference does) because that would have caused problems for common use cases (§4.3). Consider, we can:

- have auto and template argument deduction be the same for smart references
- have a smart reference deduce as a smart reference for argument passing
- have smart references and built-in references behave the same for deduction

We don't see how we could have all three. If anyone has a conclusive argument either way, we would be most responsive. We consider it essential that a smart reference passed as a template argument is passed as a smart reference because otherwise template functions and ordinary functions will be dramatically different.

6.5 More examples

More reasonably terse and reasonably real-world examples are welcome.

8 Acknowledgements

Thanks to Herb Sutter and Ville Voutilainen for discussions of **operator.()** ideas, to the CWG for finding an inconsistency, and to Holger Grund and Jeff Zhuang for constructive comments on a draft of this paper.

9 References

[Adcock,1990] James Adcock: ***Request for Consideration - Overloadable Unary operator.()***. http://www.openstd.org/jtc1/sc22/wg21/docs/papers/1990/WG21%201990/X3J16_90%20WG21%20Request%20for%20Consideration%20-%20Overloadable%20Unary%20operator.pdf

[K&S,1991] A. Koenig and B. Stroustrup: ***Analysis of Overloaded operator.()***. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1991/WG21%201991/X3J16_91-0121%20WG21_N0054.pdf

[Stroustrup,1994] B. Stroustrup: ***The Design and Evolution of C++***. Addison-Wesley.

[Powell,2004] G. Powell, D. Gregor, and J. Jarvi: ***Overloading Operator.() & Operator.*()***. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1671.pdf

[Stroustrup,2000] B. Stroustrup: ***Wrapping C++ Member Function Calls***. The C++ Report. June 2000, Vol 12/No 6.

[Stroustrup,2014] B. Stroustrup: ***Default Comparisons***. N4175.