

Operator Dot

Bjarne Stroustrup (bs@ms.com)

Gabriel Dos Reis (gdr@microsoft.com)

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 by default every operation on a **Ref** object is forwarded to the result of **operator.()**. However, an operation explicitly declared as a member of **Ref** is applied to the **Ref** object without forwarding. A programmer can prevent a pointer to a referred-to object from “leaking” into the larger program by declaring **operator.()** private.

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(int a) :p{new X{a}} {}
    X& operator.() { /* maybe some code here */ return *p; }
    ~Ref() { delete p; }
    void rebind(X* pp) { delete p; p=pp; }
    // ...
private:
    X* p;
};

Ref<X> x {99};
x.f();           // means (x.operator.()).f() means (*x.p).f()
x = X{9};       // means x.operator.() = X{9} means (*x.p)=X{9}

```

Now **Ref<X>** is a proxy for an **X** object on the free store: The **Ref<X>** behaves like an **X** object, yet does not expose pointer-like behavior to its users.

However, is that assignment 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.rebind(p);    // delete old x.p and make x.p=p – Must not forward
```

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, 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 could not define **rebind()** on the smart reference and also always have it forwarded. 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., §4.8, §6).

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.” 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).

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 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 four 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?” This design gives the designer of a handle the choice, the opportunity to prevent pointer leaks, and to return 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 value, but the assignment can be made to apply to handles or be deleted.

- **rebind()** – does a named function apply to the handle or the value? This design makes an explicitly declared handle member functions apply to the handle and all others apply to the value.

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)
{
    if (x.operator!=(y))           // hope that != is a member
        x.operator=(y.operator++()); // hope that ++ is a member
}
```

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. 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.()**.

4.2 Who is in control?

The control of whether a function **f()** is applied to the handle or 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: if the user explicitly declares a handle member it is an operation on the handle. Otherwise (even for implicitly defined operations, except destructors), 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. 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 from **(&x)->f()** was quite confusing.

4.3 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

We consider option 1 (“handle priority”) the right choice in most cases and the simplest. 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 we 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 tricky. It opens the field for all sorts of cleverness and still suffers from the problem of potentially delaying answers until template instantiation time.

Note that the priority is simply “has a member of that name been explicitly 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.

4.4 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 its matching constructor and the constructor does not forward.

4.5 Recursive use of operator.()

Is `Ref<X>::operator.()` applied to uses of a `Ref<X>` within members of class `Ref<X>`? Consider

```
template<class X>
class Ref {
public:
    Ref(int a) :p{new X{a}} {}
    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; } //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)
- implementers of “smart reference” classes are likely to be relatively few and relatively expert
- 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:
    Ref(int a) :p{new X{a}} {}
    X& operator.() { /* ... */ return *p; }
    Ref(const Ref& a) { p = (&a)->clone(); }
    Ref(Ref&& a) : p{(&a)->p} { (&a)->p=nullptr; }
    // ...
private:
    X* p;
};
```

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:
    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`.

The default copy and move operations are most unlikely to do what a programmer wants. That is unlikely to become a serious problem because their default generation will be suppressed by the presence of a declared destructor or a copy-or-move operation. However, we propose to suppress default generation of operations when an `operator.()` is declared. This would handle the unlikely case where

- a handle has no declared destructor, and
- a handle has no declared copy-or-move operation, and
- a generated default copy-or-move operation is used, and
- that operation does not result in an error

4.6 Conversions

Can a `Ref<X>` be implicitly converted to an `X` or an `X&`? For example:

```

void foo(X& x);

void bar(Ref<X> x)    // assume that Ref's copy ctor copies the handle
{
    foo(x); // does this work?
}

```

No, that would imply loss of the control of access that the definition of `operator.()` was to give.

4.7 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 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.operator.();    // x = s.a
s.operator.() = 9;      // s.a = 9

```

By itself this is useless. However, consider the proposals to allow **x.f()** to invoke **f(x)** (Glassborow,2007], [Sutter,2014], [Stroustrup,2014]), generalizing what we already do for operators, such as **==**, and for **begin()** in a range-**for**. 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.

Operators work the same way:

```

++s;    // ++s.operator.() that is ++s.a

```

The rules for handling the return type of **operator.()** are easily specified and implemented and the **x.f()** to **f(x)** transformation has precedence, but why bother? What new and useful does it provide?

Without the **x.f()** to **f(x)** transformation rule, the programmer could at most specify the transformation to use a non-member function for specifically hand-coded named functions, defeating the purpose of **operator.()**. With the transformation, static, free-standing, C-style functions can be called. In other words, relaxing the rule for what **operator.()** can return allows forwarding to a C-style interface. For example,

```

class File_handle {
    FILE* operator.() { return fp?fp:throw X{}; }
    FILE* fp = nullptr;
public:
    File_handle(const string s);
    // ...
};

File_handle fh {"myfile"};
fh.fprintf("Hello");    // fprintf(fh.operator.(),"Hello");
fprintf(fh, "World");   // calls operator.() for fh?

```


That last call is accepted if the **f(x,y)** to **x.f(y)** transformation is accepted as a general rule ([Stroustrup,2014]) and not if it is not (e.g., [Sutter,2014]).

As for **operator->()**, I propose to diagnose such an error only if the **operator.()** is actually used.

In this example, as in many, we could try the traditional solution of providing an implicit conversion from **File_handle** to **File***. However, that would allow the **File*** (the representation) to escape and could all too easily lead to surprises. The solution of providing an implicit conversion gets even more problematic when the representation is one that is easily misinterpreted. For example, implicitly “leaking” an integer supposedly representing a POSIX file descriptor would not be a good idea. When using **operator.()** to map from the handle to the representation, we carefully control where the conversion occurs.

4.8 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(); 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()
}

```

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

4.9 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 in the union of the scopes of **T1** and **T2** extended to apply to data members. This (among other things) allows us to have a simple proxy to an "object" composed of several separately allocated parts.

4.10 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
}
```

5 Restricting access

An **operator.()** can be used to gain a “naked pointer” to the object referred to. For example:

```
Ref<T> p {new T};
T* q = &p.operator.(); // pointer to “contained object”
```

This is exactly equivalent to what can be done with a smart pointer’s **operator->()**.

What if we would like to protect an object from raw pointer access? What if we want to ensure that the object “owned” by a smart reference (or smart pointer) is accessed only through that smart reference (or smart pointer)?

Unless we provide a specific **get()** operation on **operator.()** (as is popular with smart pointers), only an explicit call of **operator.()** can “leak” the pointer to the **X** in a **Ref<X>**: **x.m** always go to a member of **X** and if **X** doesn’t expose data members, we can’t even get a pointer into that **X**. So what we would like is a way to allow implicit use of **operator.()** in **x.m** while disallowing explicit use in **x.operator.()**. Consider:

```
template<typename T>
class Ref {
public:
    not explicit T& operator.();          // explicit call not allowed
    // ...
};

Ref<X> x {new X};
x.f();                                  // OK if X has an f
X* p = &x.operator.();                  // error: explicit call not allowed
```

This is syntactically unambiguous and semantically right, but we suspect that many C++ users and many C++ language lawyers would not be amused by the **not explicit (!explicit)** notation. It seems a good idea to find a simpler way to say “an explicit call of **operator.()** is not allowed.”

We already have one: **public/protected/private** access control applies to **operator.()** (as for all other members, including **operator->()**). Make **operator.()** private, and it becomes useless (to functions outside **Ref**):

```

template<typename T>
class Ref {
    T& operator.();           // explicit call not allowed (except for members of Ref)
public:
    // ...
};

Ref<X> x;
x.f();                      // error: operator.() is private
X* p = &x.operator.();     // error: operator.() is private

```

That protects the pointer to the referred-to object so well that it becomes useless. We propose an addition to the rules: access is not checked for **operator.()** and **operator->()** when they are used implicitly. The resolution then becomes:

```

Ref<X> x {new X};
x.f();                      // OK: operator.() used through .
X* p = &x.operator.();     // error: operator.() is private

```

This basically makes all implicit uses of **operator.()** work as if it was invoked by a member or a friend. The access rules for the value are unchanged. This mild abuse of the access rules seems (to us) preferable to introducing new syntax. People who are not concerned by exposing a pointer to a value object need never know. We have never seen a real-world use of access control for an **operator->()**.

This use of access control to control the use of **operator.()** is equivalent to what is done with **explicit** constructors. An alternative would be simply to ban explicit calls of **operator.()** and **operator->()**, but we don't know if there are real uses of explicit calls of **operator->()** "out there."

In C, dot is the only fundamental access operator for a **struct**. Pointer access is defined in terms of dot: **p->m == (*p).m** and to get a pointer to a variable **x**, we use **&**, and **(&x)->m == x.m**. We would like to be able to define **->**, **.**, *****, and **&** so that these equivalences hold. With **operator.()** as defined above, we can define a matched smart pointer/reference pair that matches the behavior of built-in types:

```

class Ptr {           // smart pointer to T
    T* operator->();   // access member of T
public:
    Ref<T> operator*(); // dereference yields smart reference
    // ...
};

class Ref {          // smart reference to T
    T& operator.();   // access member of T
public:
    Ptr<T> operator&(); // address of yields smart pointer

```

```

    // ...
};

```

Making **operator->()** and **operator.()** private prevents leaks of the pointer to the value.

6 Examples

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

6.1 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:

```

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

```

Potentially less stable implementation:

```

class X {
public:
    int foo();           // nonlinear
private:
    int data;
};

int X::foo() { return data; }

```

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.

6.2 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); }
private:
    T& operator.() { return val; } // don't leak a pointer to val
    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,2014a]. However, this is an example of a general technique, rather than a recommendation of how to do operator functions.

6.3 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
    // ...
private:
    T& operator.() { if (!available) read(); return obj; }
    T& obj;
    bool available; // a local copy is available through obj
    string name;
};

```

6.4 Optional

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

```

template<typename T>
class Optional {
    T& operator.() { 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 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}};
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}; })

```

Using **Optional<X>** exactly as an **X** is possible, but would probably lead to too many **obj_empty()** tests, therefore **obj_empty()** is 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.

Note that when using **auto**, we extract a reference to the contained object (the value). This can be important for avoiding repeated tests. The choice between having = return a handle or a value is an important design point.

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>**.

6.5 More examples

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

7 Working paper text (drafty)

Add a paragraph to 13.5.6 (the current first and only paragraph is repeated here to provide context):

13.5.6 Class member access [over.ref]

operator-> shall be a non-**static** member function taking no parameters. It implements the class member access syntax that uses **->**.

```

postfix-expression -> templateopt id-expression
postfix-expression -> pseudo-destructor-name

```

An expression **x->m** is interpreted as **(x.operator->())->m** for a class object **x** of type **T** if **T::operator->()** exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3).

No access control is applied to **operator->()** when invoked through the **->** operator.

operator. shall be a non-**static** member function taking no parameters. It implements the class member access that is not through a pointer, whether the syntax that uses **.** or not.

```

postfix-expression . templateopt id-expression
postfix-expression . pseudo-destructor-name

```

Unless **m** is explicitly declared to be a **public** member of **x**'s class or the destructor, the expression **x.m** is interpreted as **(x.operator.()).m** for a class object **x** of type **T** if **T::operator.()** exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3). Unary and binary operators are interpreted as calls of their appropriate operator functions (???) so that the previous rule apply. [[for example, **x=y** is interpreted as **x.operator=(y)** which is interpreted as **x.operator.().operator=(y)**.]] Implicit or explicit destructor invocations do not invoke **operator.()**.

No access control is applied to **operator. ()** when invoked through the **.** operator or implicitly for some other operator.

TBD: Wording for “a **operator.()** declaration suppresses the generation of destructor, copy operations, and move operations.”

TBD: Do we need text in the expressions section? In particular, do we need to say something to stope the compiler from rewriting **p->m** to **(*p).m** and then apply **operator.()**?

8 Acknowledgements

Thanks to Herb Sutter for discussions of **operator.()** ideas. Thanks 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.openstd.org/jtc1/sc22/wg21/docs/papers/1991/WG21%201991/X3J16_91-0121%20WG21_N0054.pdf
- [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,1994] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley.
- [Stroustrup,2000] B. Stroustrup: *Wrapping C++ Member Function Calls*.
The C++ Report. June 2000, Vol 12/No 6.
- [Stroustrup,2014] B. Stroustrup: *Call syntax: x.f(y) vs. f(x,y)*. N4174.
- [Stroustrup,2014a] B. Stroustrup: *Default Comparisons*. N4175.