Dmitry Lenkov
HP California Language Lab
19447 Pruneridge Avenue MS 47 LE
Cupertino CA 95014

William M. Miller
GLockenspiel Ltd.
PO Box 366
SudBury, MA 01776-0003

Oct 8, 1990

Request for Consideration: Overloadable Unary operator.()

I humbly request the C++ standardization committee consider allowing overloadable operator.(), operator to work analogous to overloaded operator->().

Discussion as follows:

With few exceptions, C++ allows all its operators be overloaded. The few exceptions are: . [dot,] .* [dot star,] :: [colon colon,] and ?: [binary selection.] The commentary on page 330 ARM, gives the following "explanation" :

The reason for disallowing the overloading of ., .*, and :: is that they already have a predefined meaning for objects of any class as their first operand. Overloading of ?: simply didn't seem worthwhile.

I agree I can't see any worthwhile reason for overloading ?:, but the reason given for disallowing the other operators cannot hold, because there is already a counterexample: unary operator& already had a predefined meaning, yet it is overloadable.

Three questions to be answered in considering operator.() as a candidate for overloading are: 1) would doing so cause any great problem? 2) are there any compelling reasons to allow it? 3) what overloadings of operator.() should be permitted? I claim these questions can be easily answered as follows: 1) allowing operator.() to be overloaded causes no great problems. 2) there are compelling reasons to allow it and 3) unary operator overloading analogous to what is permitted of unary operator->() should be permitted. IE unary member overloaded operator.(), which can be called with an object or reference of the class it is defined in, or derived class, on the left hand side, returning a reference or object of a class to which . can be applied again.

Discussion of these claims:

"Allowing operator.() to be overloaded causes no great problems."

Unary operator& demonstrates that there is no problem overloading a function for which there is already a pre-defined meaning. The implementation of operator.() in other respects is similar to operator->() which also has been successfully implemented by several compilers, demonstrating that no new technology is required to implement unary operator.(). Like operator&, and operator->(), operator.() is never invoked except on a lhs object of a class explicitly overloading operator.(), thus no existing code can be affected by this change.

Some people have expressed concern that if operator.() is overloadable, then how does one specify member selection of that class members themselves necessary to implement operator.() ? In practice, this does not prove to be a problem. Operator.() is invoked only in situations where . [dot] is explicitly used, and when writing smart reference classes, proxies, and other simple classes one typically accesses members via "implied this->", thus one doesn't use . [dot]. In situations where one would normally use . [dot], such as when a class instance is passed as a parameter to a member function, getting an overloaded operator.() can be sidestepped via pointer syntax: use (&ob)->member, rather than ob.member. People next complain that this means it will be difficult to simultaneously overload operator->() and operator.() to which I reply: Thank God! We don't need classes that try to act simultaneously as pointers and references! That's the whole point of allowing operator.() to be overloaded: so that objects that act like pointers can use pointer syntax, and objects that act like references can follow reference syntax!

"There are compelling reasons to allow it"

Overloading operator.() is necessary in practice to allow "smart reference" classes similar to the "smart pointer" classes permitted by overloading operator->(). Overloading operator->() to access objects following reference semantics is pretty workable:

```
RefCntPtr pob;

pob = pobOther;
pob->DoThis();
pob->DoThat();
```

However, if the object needs to follow value semantics, then this solution becomes onerous:

```
RefCntHugeIntPtr pA, pB, pC;
//....
*pC = *pA + *pB;
int digit104 = (*pC)[104];
pC->truncateNdigits(100);
```

What you really want to be able to do for objects that require value semantics is create smart references as follows:

```
RefCntHugeIntRef a, b, c;
/....
c = a + b
int digit104 = c[104];
c.truncateNDigits(100);
```

Another common case where you'd rather have overloaded operator.() rather than operator->() is in creating proxy classes. The proxy class just forwards messages to its destination classes. A proxy class can be used in many ways. An example is a proxy member, allowing a runtime decision of the actual implementation of that member. Or a class can even inherit from a proxy, allowing the behavior of its parent be specified at run.
[Behavior, but not protocol, that is] Of course, pointer syntax can be consistently used for
these proxy cases, but the underlying implication is that object instances are being created dynamically on the heap, not statically, nor on the stack.

Note, that at a relatively high leve of pain, classes obeying reference semantics can already be created: One simply writes a class that contains a pointer that can be assigned to the forwarding object, and write an inline forwarder function for each and every member function in the protocol:

```
class FooProxy
{
    foo* pfoo;
public:
    FooProxy(foo* pfooT) : pfoo(pfooT) {}
    void DoThis() { pfoo -> DoThis(); }
    void DoThat(int i) { pfoo -> DoThat(i); }
    int ReturnInt() { pfoo -> ReturnInt(); }
//  ....
    void NthMemberOfProtocol() {pfoo ->
NthMemberOfProtocol();}
};
```

Needless to say, when most class writers are faced with the prospect of manually writing a forwarding function for each and every function in a protocol, they don't! They punt instead, and overload operator->(), even when the rest of their class follows value semantics. Thus, class users end up having to guess whether to use . or -> as the member selector in every context. If operator.() is overloadable as well as operator->(), then customers can learn the simple convention: "Use . whenever dealing with object obeying value semantics, use -> whenever dealing with objects obeying reference semantics."
In short, lacking operator.(), class writers are forced to violate convention meaning of operator->(). Instead, we should enable a complete set of overloadable operators, so that class writers can maintain historical meanings and usages of these operators.

I therefore ask due consideration be given to allowing operator.() to be overloaded analogous to operator->(). I suspect that the committee should then consider also whether operator.*() be overloadable analogous to operator->*(). However, I am not asking for that, since I do not consider myself sufficiently experienced with member pointers to be aware of the ramifications. Let someone else propose the necessary changes for operator.*(), if they so choose.

The necessary changes to the Annotated Reference Manual to support operator.() are listed below. I list the changes necessary in ARM, rather than the product reference manual, since the changes necessary to ARM are a pure superset of the changes necessary to the product reference manual.

Jim Adcock, Oct. 8, 1990

Jim

Section 7.2.1c

Compiler vendors would need to add a convention for encoding operator. [dot.] But this is not an issue for the standardization effort.

Section 12.3c

Add the following table entry:

. [operator dot]	yes	yes	yes	member	no

Chapter 13, page 307, line 6, change to:

and unary class member accessors -> and . [dot]) when at least one operand is a class object.

Page 330:

operator: one of .... -- add . [dot] to the list

The following operators cannot be overloaded: .... remove . [dot] from the list.

The reason for disallowing the overloading of ., .*, and :: is that they already have a predefined meaning for objects of any class as their first operand. Overloading of ?: simply didn't seem worthwhile.

Change To:

The reason for disallowing the overloading of :: and ?: is that it simply didn't seem worthwhile.

Section 13.4.6

Add the following text:

Class member access using . [dot]

primary-expression . primary-expression

is considered a unary operator. An expression *x.m* is interpreted as *(x.operator.()).m* for a class object *x*. It follows that *operator.()* must return either a reference to a class or an object of or a reference to a class for which *operator.()* is defined. *operator.()* must be a nonstatic member function.

Commentary:

Note that Ellis and Stroustrup's annotated notes on page 337 could have been just as well written as follows:

Consider creating classes of object intended to behave like what one might call "smart references" -- references that do some additional work, like updating a use counter on each access through them.

```
struct Y { int m; };

class Yref {
    Y* p;
    // information
public:
    Yref(const char* arg);
    Y& operator.();
};

Yref::Yref(const char* arg)
{
    p = 0;
    // store away information
    // based on 'arg'
}

Y& Yref::operator.()
{
    if (p) {
        // check p
        // update information
    }
    else {
        // initialize p using information
    }
    return *p;
}
```

Class Yref's . [dot] operator could be used as follows:

```
void f(Yref y, Yref& yr, Yref* yp)
{
    int i = y.m;          // y.operator.().m
    i = yr.m;             // yr.operator.().m
    i = yp.m;             // error: Yref does not have a member m
}
```

Class member access is a unary operator. An operator.() must return something that can be used as an object or reference.

Note that there is nothing special about the binary operator .* [dot star.] The rules in this section apply only to . [dot].

End Commentary.

Thank you for your consideration, and please inform me of your decisions.

James L. Adcock
Microsoft
One Microsoft Way
Redmond, WA 98052-6399