Overloading operator.() & operator.*()

Gary Powell, Doug Gregor, Jaakko Järvi

Project: Programming Language C++, Evolution Working Group
Reply-to: Gary Powell <powellg@amazon.com>

Abstract

With operator->() and operator->*() a designer of a class can create a smart pointer. There is also a need to create smart references, and smart delayed member access variable hence the need to be able to overload operator.() and operator.*().

1. Background

The ability to overload operator.() and operator.*() has been discussed for a number of years. Our understanding is that this operator was heavily discussed at the meeting in London and not accepted at that time. We have included several emails, one of which has a copy of the original proposal, so that newer members of the committee can review some of the historical issues. There were many more comments made at the time and interested readers should have no trouble finding them using these three links as a beginning reference.

In 1992, Fergus J. Henderson did an implentation of operator.() overloading for gcc versions 2.3.x and 2.4 based on Jim Adcock's proposal, proving that it could be done.

We believe that since 12 years have passed there has been enough work done in C++ to reopen the topic and have a rational discussion of the technical issues. There are some techniques that have been learned that can overcome some of the technical problems that were originally identified by reviewers of the original proposal. Especially in light of all the metaprograming and template work that has been done, there are now stronger cases to allow overloading of these operators. These newer libraries have shown us that the addition of operator.() and operator.*() could benefit all users of C++, by allowing the creation of smart references and proxy classes that behave in a natural way.

Creation of a smart reference class gives us the ability to deference a smart pointer giving the class control of the object rather than exposing the underlying data, or preventing apparent direct access to it.

There were several objections raised to allowing overloading operator.() and operator.*():

- creating write only code
- difficult access to member functions
- operators such as "+" and "=" won't have the right semantics

We believe that recently discovered techniques allow us to overcome those objections and write reasonable and readable code.

- Writing clear code is a general technique not specific to a particular feature of the language.
- There are a number of ways to access the member functions which we will show in detail.
- Overloading operators can be in part done automaticly and with some manual work so that the semantics are correct.

We believe that not being able to overload operator.() and operator.*() is an unnecessary wart in the language. The operator->() and operator->*() act as users of the class expect but operator.() and operator.*() do not. This paper proposes that we fix this language feature in way that makes usage of proxy classes easier and more natural.

2. Problems to be Solved

The general need for operator.() arises from creating a proxy class which has the same semantics as the object it proxies. Without operator.(), class designers instead create a class that appears to be a pointer container when in fact it may be an object container. This is because we can overload operator->() but not operator.().

In general designers of a class which use the operator.() and operator.*() will have only a few public access named functions. The design of the class is to closely mimic the class it represents in all use cases. The class created will act as a proxy to the underlying data and forward all requests for member access to the object being proxied.

There is at least one member function that comes to mind that would likely be a member of the proxy class. In particular there has been a suggestion that a rebind member function will be desired by users of a smart reference class. The more general use case the class will have only overloaded operators, constructors and destructors. This is to prevent user confusion with calling named functions as access to them must be made explicit. Or will be through a non overloaded address operator, and pointer operator, (&A)->memfn().

The main reason for having an operator.() is to compliment a smart pointer with a smart reference. The operator*() of the smart pointer class returns a smart reference. The operator&() of the smart reference class returns a smart pointer. By overloading the operator.() of the smart reference we allow access to the underlying member functions of the object being proxied by these two classes.

There are a number of proxy classes, handle-based memory, resource managers, etc. that would benefit from this set of operator overloads, and rather than being more confusing to use, would be less confusing. And yet the owner of the library could continue to maintain control and access to the object. Naive users expect that the operator*() on a smart pointer would return an object through which one could access the member functions via the operator.().

2.1. Creating smart references, proxy objects

The following code is a simple example of a smart reference class.

EXAMPLE

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


class SimpleSmartReference {
public:
   SimpleSmartReference(T &rhs) : t(rhs) {}
   T &operator.() { return t; }
   T const &operator.() const { return t; }
private:
   T &t;
};


SimpleSmartReference x1;
x1.f();              // call T::f();
```

```
SimpleSmartReference const cx1;
cx1.f();  // call T::f() const;
```

As you can see, all calls to the members of T are forwarded through operator.().

2.2. Allowing dereferencing of smart pointers.

One of the problems with a smart pointer class is what about operator*()? Do we return a reference to the underlying object and lose whatever control we have over the object? Or do we remove this from the user's grasp by making it private?

With a smart reference we can give users a safe object through which they have apparent direct access to the object.

Here is a simple example to illustrate the problem and the solution we would like to have.

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

template<typename T>
class SmartReference;

template<typename T>
class SmartPointer {
public:
   SmartPointer(SmartReference<T> &rhs);
   T *operator->() { return t; }
   T const *operator->() const { return t; }
   SmartReference<T> operator*() { return SmartReference<T>(*t); }
   SmartReference<T> const operator*() const { return SmartReference<T>(*t); }
private:
   T *t;
};

template<typename T>
class SmartReference {
public:
   SmartReference(T &rhs) : t(&rhs) {}
   T &operator.() { return *t; }
   T const &operator.() const { return *t; }
   SmartPointer<T> operator&() { return SmartPointer<T>(&t); }
   SmartPointer<T> const operator&() const { return SmartPointer<T>(&t); }
private:
   T *t;
};
```

```
SmartPointer<T> sp(new T);
sp->f()             // calls T::f()
SmartReference<T> sr = *sp;
sr.f()              // calls T::f()

SmartPointer<T> const spc(new T);
spc->f()            // calls T::f() const
SmartReference<T> const sr = *spc;
src.f()             // calls T::f() const
```

2.3. Fixing Language Inconsistency

In the original proposal the following reason was given for fixing this language inconsistency. [Adcock July 1990]

> 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."

We agree in general, however any discussion about overloading ?: is out of scope for this proposal. The reason given for disallowing the operator.() and operator.*() has a counter example: unary operator&, which already had a predefined meaning, yet it is overloadable.

3. Proposal

We are proposing that operator.() behave exactly like operator->(). For each object that has an operator.() overload that an object must be returned that has a member function or member variable with the same name as that used on the right side of operator.(). This is analogous to operator->() which requires a pointer to be returned which has a member function or member variable with the same name as that used on the right side of operator->().

3.1. Example

```
template <typename T>
class SmartReference {
public:
   SmartReference(T &rhs) : t(rhs) {}
   T &operator.() { return t; }
   T const &operator.() const { return t; }
   void f();
   void f() const;
   void g() {
      ::f();                  // call global scope f();
      (*this).f();            // call T::f();
      f();                    // call SmartReference::f();
      SmartReference::f();    // call SmartReference::f();
      this->f();              // call SmartReference::f();
   }
   void g() const {
      ::f();                  // call global scope f();
      (*this).f();            // call T::f() const;
      f();                    // call SmartReference::f() const;
      SmartReference::f();    // call SmartReference::f() const;
      this->f();              // call SmartReference::f() const;
   }
private:
   T &t;
};


SmartReference<T> x;
x.f();                        // call T::f();
(&x)->f();                    // call SmartReference::f();


SmartReference<T> const cx;
cx.f();                       // call T::f() const;
(&cx)->f();                   // call T::f() const;
```

3.2. Extensions

We are proposing a relatively minor extension to the language definition that has major implications, and yet has major benefits to writers and users of proxy classes. The proposal is that operator.() and operator.*() behave similarly to operator->() and operator->*().

3.3. Standardese Changes

REMOVE

13.5.3 The following operators cannot be overloaded:

"." and ".*"

INSERT

13.5.1 To the table of overloadable operators, add "." and ".*"

13.5.6.2 (Class member access) operator. shall be a non-static member function taking no parameters. It implements class member access using .

postfix-expression . id-expression

An expression x.m is interpreted as (x.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 overloaded resolution mechanism(13.3).

13.6.26 For every quintuple *(C1, C2, T, CV1, CV2)* , where *C2* is a class type, *C1* is the same type as *C2* or is a derived class of *C2*, *T* is an object type or a function type, and *CV1* and *CV2* are cv-qualifi er-segs, there exist candidate operator functions of the form

CV12 T& operator.(CV1 C1, CV2 T C2::*);*

where *CV12* is a union of *CV1* and *CV2.*

Annex A Grammar summary A.11 amend to include "." and ".*"

4. Objections to this Proposal

4.1. Objection 1: Write only code

The primary objections have been how to educate users of the syntax for getting at the actual un-overloaded operator.() and that it would be possible to create write only code.

With the passage of time it has become clear that while designers of a class can always create write only code by misusing operator overloading, in fact few do. Novice users steer clear of operator overloading when creating a class because of the obvious diffi culties. Expert users are often left frustrated at their inability to create a class which behaves in a "natural" way. And novice users of those classes are wondering why they have a convoluted syntax to what should be a simple operation.

4.2. Objection 2: Access to member functions is difficult

When a class has operator.() overloaded, access to members of that class is now more diffi cult because all operations which use operator.() now pass through to the underlying object. There are a number of techniques to deal with this problem and we will address them below.

4.2.1. Direct access within the class

Within the class itself there is no problem at all with accessing the member functions and data. Calls to the members do not pass through operator.() as there is an implicit call to this->memberFn(), or this->memberData. This is analogous to overloading operator->() where calls to this->memberFn() do not pass through the overloaded operator unless explicitly called out.

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

```
template <typename T>
class SmartReference {
private:
   T t;
   void f();
public:
   ...
   void f();
   void g() {
      f();          // call SmartReference::f()
      this->f();    // call SmartReference::f()
      (*this).f();  // call T::f()
      t.f();        // call T::f()
   }
};
```

As you can see the common calls to qualfi ed and unqualifi ed function f(), behave exactly as the designer of the class would expect.

4.2.2. Use operator&()

For users of the proxy class there may still be a need to call member functions of that class. One way is to use operator&() and operator->(), which bypass the call to operator.()

```
template <typename T>
class SmartReference {
   T *t;
public:
   ...
   void rebind(T*);
};

SmartReference<T> x(new T);
(&x)->rebind(new T);
```

This is ugly but readable.

For some smart reference classes it may be desired to overload the operator&(). This problem can lead to an inability to call any named functions of the smart reference class directly.

4.2.3. Use operator.*()

One technique to bypass the overload of address & is to use operator.*() instead.

```
SmartReference<T> sr(sp1);

(sr.*&SmartReference<T>::rebind)(sp2);
```

This is ugly but works unless we have overloaded operator.*()!

Using a templated function we can hide some of this machinery.

```
template<typename T1, typename T2>
void rebind(SmartReference<T1> &sr, SmartPointer<T2> &sp) {
  (sr.*&SmartReference<T1>::rebind)(sp);
}
```

Then users would call

```
rebind(sr, sp2);  // not too ugly but obscure.
```

4.2.4. Use addressof

Using a technique used in boost, to always get the address of an object, even if it has overloaded operator&(), called "addressof" you can bypass the call to SmartReference::operator&(). The core code for addressof is from <boost/utility/addressof.hpp> is copied here:

```
template <typename T>
T* addressof(T& v) {
 return reinterpret_cast<T*>(
     &const_cast<char&>(reinterpret_cast<const volatile char &>(v)));
}
```

Thus one can call a member function of a SmartReference using this technique.

```
SmartPointer<T>          sp1;
SmartReference<T>        sr(sp1);
SmartPointer<T>          sp2;

(addressof(sr))->rebind(sp2);        // also ugly but works.
```

4.2.5. Use Inheritence

It has been suggested that another way around the **(&r)->** syntax is to use inheritence for the member functions which must be named. This way the user can cast the reference to its base class and access the necessary member functions.

```
class SmartReferenceMemberFns {
protected:
   Num *p;
public:
   rebind(Num &rhs) { p = &rhs; }
};

class SmartReference : public SmartRefernceMemberFns {
public:
   ...
};
```

Users of this smart reference class would access the member functions like this.

```
Num x;
SmartReference r(x);
SmartReferenceMemberFns &r_member_fns = r;
Num y;
r_member_fns.rebind(y);
```

This obviously isn't perfect either but its not totally unreasonable or unreadable. Good class and function naming choices can help the readability a lot with this technique.

In general calls to member functions would not be the most common usage pattern for a smart reference. Thus having a slightly more cumbersome interface is a reasonable trade-off.

4.3. Objection 3: operator+(), and operator=() won't have the right semantics

Some of the other ideas about the possible uses of operator.() come from *"The Design and Evolution of C++"* section **11.5.2** where a smart reference is used on a user's large digit number class, Num. We'll repeat the example here for those who do not have a copy of the **D&E** close at hand.

```
class Num {
  // ...
public:
  Num &operator=(const Num&);
  int operator[](int);                  // extract digit
  Num operator+(const Num &);
  Num &operator++();
  void truncateNdigits(int);            // truncate.
};
```

Where it is desired to defi ne a class RefNum that behaves like a Num& except for performing some added actions. For example,

```
void f(Num a, Num b, Num &c, int i) {
  c = a + b;
  int digit = c[i];
  c.truncateNdigits(i);
}
```

then we would be able to write a function that takes a RefNum the same way we use a Num:

```
void g(RefNum a, RefNum b, RefNum c, int) {
 c = a + b;
 int digits = c[i];
 c.truncateNdigits(i);
}
```

If we defi ne the class RefNum as:

```
class RefNum {
   Num *p;
public:
   RefNum(Num &rhs) : p(&rhs) {};
   Num & operator.() { return *p; }
   void rebind(Num &rhs);
};
```

We don't get the effect desired because "." isn explicitly mentioned in all cases.

```
c = a + b;                  // no dot
int digits = c[i];          // no dot
c.truncateNdigits(i);       // call operator.()
```

4.3.1. Write explicit forwarding

To get around this problem we could write the necessary forwarding functions.

```
class RefNum1 {
   Num *p;
public:
   RefNum1(Num &rhs) : p(&rhs) {};
   Num & operator.() { return *p; }
   void rebind(Num &rhs);

   // forwarding functions;
   RefNum1 &operator++() { ++(*p); return *this; }
   RefNum1 & operator=(const RefNum1 & rhs) { *p = *rhs.p; return *this; }
   RefNum1 operator+(const RefNum1 &rhs) { return (*p + *rhs.p); }
   ...
};
```

As Bjarne notes this is tedious, but is possible as the set of overloaded operators is a finite and bounded group.

4.3.2. Use enable_if<>

With the discovery of the technique, Signature Failure Is Not An Error, or SFINAE, and using the enable_if<> template and some other template techniques a general purpose forwarding template could be written once for each operator overload to auto detect and generate the necessary forwarding function.

4.3.2.1. Automatic operator detection

```
// detectIncrementOperator.h (c) June 2003 Mat Marcus, Jeremy Siek, Jaakko Järvi, Dietmar Kuehl

struct bad_conversion
{ template<typename T> bad_conversion(const T&); };

// Indicates the lack of an increment operator
struct no_increment_op { char make_bigger[32767]; };

// Fallback operator++, if no other operator++ is better
no_increment_op operator++(bad_conversion const &);

// Magic type
struct checking_incrementable {};

// tag types that indicate the result
typedef char (&yes_type)[1];
typedef char (&no_type)[2];

template<typename T>
yes_type operator,(const T&, checking_incrementable);

no_type operator,(no_increment_op, checking_incrementable);
```

```
template<typename T>
struct is_incrementable {
   static T t;
   static const bool value = sizeof(++t, checking_incrementable() ) != sizeof(no_type);
};

class RefNum2 {
   Num *p;
public:
   RefNum2(Num &rhs) : p(&rhs) {};
   Num & operator.() { return *p; }
   void rebind(Num &rhs) { p = &rhs; }

   // forwarding function if they exist for type Num;
   enable_if<is_incrementable<Num>, RefNum2 &>::type operator++()
      { ++(*p); return *this; }
   ...
};
```

However this technique does not work for a few operators, specifically:

- operator[](T&)
- operator=()
- any conversion operators

These operators must be members of the class rather than being defined in the global namespace.

4.3.3. Combine the Two Techniques

To make an exact proxy class we are left with implementing some of the operators and inheriting from a smart reference class which has auto detection of all of the operators for which that is possible.

```
class RefNum : public SmartReference<Num> {
public:
   RefNum(Num &rhs) : SmartReference<Num>(rhs) {};
   RefNum &operator=(RefNum const &);
   Num operator[](int);
   void rebind(Num &rhs);
   ...
};
```

This is a very managable task and perhaps with policy classes could be even easier.

4.3.4. Metaprogramming futures

There have been a couple of unofficial proposals floating around that would provide the programmer with information that the compiler knows but the programmer cannot get to during the compilation of the code. With a metaprogramming language the information about the overloaded operators could become available to the library writer. Thus intelligent choices could be made and only the necessary operations would be created. And information about the class being proxied would be available which could be used to make an even smarter proxy class.

While a Meta Programming Langauge would be nice, it is not essential to this proposal.

5. Alternative proposals

There have been several alternative proposals for when to forward operator.() which we will discuss.

5.1. Implicit operator.()

5.1.1. Apply it all the time

One other suggestion that Bjarne & Andrew Koening made to fix the forwarding problem is to apply operator.() to every operation on a RefNum. That way the original definition of RefNum would make the original example work as desired and expected.

However applying operator.() this way implies that to access a member of RefNum itself you must use a pointer.

```
void h(RefNum r, Num &x)
{
   r.rebind(x);     // error: no Num::rebind
   (&r)->rebind(x); // ok call RefNum::rebind;
}
```

Note that if we had overloaded operator-> for RefNum to forward it to the Num *, we couldn't even do this without using addressof or one of the other techniques previously discussed.

5.1.2. Apply operator.() if only there is no matching function

The same section of the **D&E** then suggests another way around the problem of implicit calls to operator.(), and that is to not forward the call to a member function unless there isn't a matching member function in the class that has operator.().

```
class Num {
public:
   ....
   rebind();
}

void h(RefNum r, Num &x) {
   r.rebind(x);  // calls RefNum::rebind not Num::rebind();
}
```

This could cause confusion among users of the class Num, because member functions they thought would be forwarded to are not. However, advanced users of C++ are used to this sort of problem as it crops up with the misuse of inheritance. This proposed new rule may also place an undue burden on compiler writers.

6. operator.*()

In the coding experience of the authors, the operator.*() is less used than other operators, and thus the overload arguments are less compelling. And therefore so are the arguments against allowing it as fewer programmers would run into it. The main use cases that are apparent to us involve Boost.Lambda and using the lambda variables.

In Boost.Lambda [Järvi & Powell] there is an overloaded operator->*(), since operator.*() is not available now, users who have objects instead of pointers must combine it with a call to operator&()

```
lambda_var x;

((&x)->*&T::memFn)();
```

vs using operator.*()

(x.*&T::memFn)();

These calls are within a greater lambda expression which is not included here for clarity.

A lambda variable in this case needs to know what member variable or function access is required and to store this information away for later use. While it would be beneficial to lambda to know the function access using operator.() and stop the forwarding, this would be so unlike the current access to operator->() which has a similar issue that we are not proposing this change.

Each of these uses is a bit more obscure and have been overshadowed by the use of Boost.Lambda's bind instead of operator->*(). However we believe for completeness operator.*() should have the same overload capabilities as operator->*(). This will be a least unexpected result vs prohibiting it and it will make lambda programs easier to maintain and read.

7. Summary

The main argument for disallowing operator.() appears to be the preceived inability to access the member functions of the class which has this operator overloaded. We have demonstrated a number of methods which would allow access through a reasonable interface and yet not be overly complex to use. There is now sufficient history and usage of proxy classes to demonstrate the need for smart references that behave as users expect.

There is also enough experence in writing complex classes by experienced software engineers to demonstrate that they are capable of making intelligent design choices given the powerful tools already at their hand. With power comes responsiblity, and we believe that in general C++ programmers can handle this additional power.

There is a desire to eliminate the drudgery of writing all of the overloaded operators necessary to do proper forwarding. We believe that we have shown that most of this drudgery can be done once. Users of operator.() can use operator detection code which will make overloading operators a managable task. Therefore, having an implicit call to operator.() would in fact cause more trouble to users of smart references than it would save writers of that class.

The need for operator.() is best characterized as the desire to be able to write a smart reference class. Smart references are the complementary class to a smart pointer class. They go hand in hand and together are more useful than each by itself.

Operator.*() is the complementary operator to operator->*() for member access to objects through a delayed operation variable i.e. a lambda variable. The use of this operator is the natural way to express access to the member funtions of an object rather than having to take its address to merely be able to use operator->*().

8. Acknowledgements

We have had many private conversations with individuals who wish to remain anonymous. We thank them anyway. All errors and omissions are of course ours.

"Design and Evolution of C++" by Bjarne Stroustrup ISBN 0-201-54330-3

Proposal for operator.() Jim Adcock & Steven Kearns
http://groups.google.com/groups?selm=1992Jul14.190540.2425%40microsoft.com&oe=UTF-8&output=gplain

J.Järvi and G. Powell. The Boost Lambda library.
http://www.boost.org/libs/lambda/doc/index.html, March 2002

9. APPENDIX A

http://groups.google.com/groups?selm=1992Jul14.190540.2425%40microsoft.com&oe=UTF-8&output=gplain
Newsgroups: comp.lang.c++
From: jimad@microsoft.com (Jim Adcock)
Subject: original [re-typed] text of overloadable operator dot proposal
Message-ID: <1992Jul14.190540.2425@microsoft.com>
Date: 14 Jul 92 19:05:40 GMT
Organization: Microsoft Corporation
Lines: 319

<intro text sniped>

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 predefi ned meaning for objects of any class as their fi rst 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 predefi ned 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 defi ned 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-defi ned 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 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 diffi cult 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!

[NOTE INSERTED BY THE AUTHORS OF THE 2004 PROPOSAL]

Jim Adcock appears to have dismissed this objection a little too quickly. The point of this objection is that people would like to be able to overload operator "&" on a smart reference to return a smart pointer, and to overload operator "->" on a smart pointer to return a smart reference. In that case, there is a need to be able to call member functions of the smart pointer or smart reference class. Jim Adcock's proposal did not address this. This proposal, however, does address this, in section 4.2

"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 to be specifi ed 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 level 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 conventional 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 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 suffi ciently experienced with member pointers to be aware of the ramifi cations. 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

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:

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 predefi ned 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 is 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 a well written as follows:

Consider creating classes of objects intended to behave like 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

====== End of original text [retyped] above ======

10. APPENDIX B

http://groups.google.com/groups?selm=23.UUL1.3%238618%40softrue.UUCP&oe=UTF-8&output=gplain

From: kearns@softrue.UUCP (Steven Kearns)
Newsgroups: comp.lang.c++
Subject: overloaded dot
Message-ID: <23.UUL1.3#8618@softrue.UUCP>
Date: 5 Oct 91 21:29:52 GMT
Organization: Software Truth
Lines: 113

This is a response to "Analysis of overloaded operator .()" ,
doc # x3j16/91-0121 in the latest c++ mailing.

The document outlines a number of problems with handling operator.()
and solicits suggestions. Here is one that stresses compatibility
with existing C++ traditions and philosophy:

For the purposes of description, lets say that for any class X which
has operator.() defi ned, X' is the same class without operator.()
defi ned.

In summary, the entire proposal is that X.foo is interpreted as
X'.operator.().foo when X has defi ned an operator.(), and that all
other C++ rules stay the same.

We start with the analogy from operator->(), and say that when X has
operator.() defi ned, then X.foo is interpreted as X'.operator.().foo.

Then, what about "X++"? While it is tempting to argue that X++ is
equivalent to X.operator++(), in fact the most reasonable defi nition
is to equate X++ with X'.operator++(). The reason for this
preference is to keep with the tradition of C++ that insists that
X->foo is not the same as (*X).foo when X has defi ned an
operator->(), and that X++ is different from "X = X+1" when X has
overloaded "++". It similarly follows that "X.operator++()" is
equivalent to "X'.operator.().operator++()", because the dot operator
was explicitly used. Also, "X = Y" is equivalent to "X'.operator=
Y".

There remains the problem of how to access members of X. I suggest
either of the following:

(a) pick a unary operator and have it return a pointer to X when applied
to X. In the following example we have used "&".

```
class X {  // acts like a reference to a T
        T * ptr;
```

```
        T& operator.() { return *ptr };
        X * operator&() { return this; }
        T * sample { return (&X)->ptr; }
};
```

(b) a more general solution is to defi ne a cast operator to an "X*":

```
class X { // acts like a reference to a T
        T * ptr;

        T& operator.() { return *ptr };
        typedef X * MakeXstar;
        operator MakeXstar() { return this; }
        T * sample { return MakeXstar(X)->ptr; }
};
```

The only disadvantage that I see of this proposal appears when trying to use operator.() to implement a class that acts like a reference. For example, defi ne IntRef to act like an int&:

```
struct IntRef {
        int * i;

        int& operator.() { return *i; }

        // here is the bummer
        int& operator++() { return (*i)++; }
        int * operator&()  { return &(*i); }
        .... etc....

        typedef IntRef * MakeIntRefStar;
        operator MakeIntRefStar() { return this; }

};
```

The bummer is that for each of the built in operators (i.e. "+", "*", "&", "/", etc..) that the underlying class understands, you must defi ne an operator in the smart reference class which does the right thing to the underlying object. This just adds to the work involved in defi ning a smart reference class, compared to some of the other proposals. However, I think the trade off is worth it since consistency is maintained.

Since different classes overload different sets of operators, my proposal probably prevents one from writing a template class for a smart reference. For example, consider replacing the "int" in IntRef with Y:

```
struct YRef {
        Y * yptr;
```

```
        Y& operator.() { return *yptr; }

        // here is the bummer
        Y& operator++() { return (*yptr)++; }
        Y * operator&()  { return &(*yptr); }
        .... etc....

        typedef YRef * MakeYStar;
        operator MakeYStar() { return this; }

};
```

The problem is that class Y may not have defi ned operator++(), and also Y::operator++() may not return a Y.

****
* Steven Kearns ....uunet!softrue!kearns *
* Software Truth softrue!kearns@uunet.uu.net *
****

11. APPENDIX C

http://groups.google.com/groups?selm=xs3ccha%40microsoft.UUCP&oe=UTF-8&output=gplain

From: jimad@microsoft.UUCP (Jim ADCOCK)
Newsgroups: comp.lang.c++
Subject: Re: overloaded dot
Message-ID: <xs3ccha@microsoft.UUCP>
Date: 11 Oct 91 17:59:48 GMT
References: <23.UUL1.3#8618@softrue.UUCP>
Organization: Microsoft Corp.
Lines: 104

In article <23.UUL1.3#8618@softrue.UUCP> kearns@softrue.UUCP (Steven Kearns) writes:
|
|This is a response to "Analysis of overloaded operator .()" ,
|doc # x3j16/91-0121 in the latest c++ mailing.
|

I apologize that since the x3j16 voted against sending out mailings
to everyone, I haven't been able to get a copy of x3j16/91-0121 yet.

Still, I went over your response to the Analysis, and you conclusions
see to be the same as I was thinking.

Here's my perspective on the situation:

Maybe a year ago I sent in a proposal for an operator dot to the x3j16
committee's two principle members, asking for consideration of the
issue, and for membership info. Since this was before the document
numbering system got started, the document never seems to have gotten
a number. Bjarne has stated that they were considering operator dot in
subcommittee, but I don't know if they were reviewing my document,
or something else. There have been a number of people over the last
year who have criticized the operator dot idea -- but none of them
seem to have read the proposal, so I'm not sure what's going on.

In any case, my proposal was very simple and short. It pointed out
that Bjarne's reasons for not allowing overloaded operator dot appeared to
be incorrect [ARM page 330], that there are situations where it would
be useful -- namely "smart reference" classes, that removing an unnecessary
special case restriction from the language against overloading operator
dot would simplify the language -- we could just say: "all operators,
unary and binary are overloadable." It went on to say that operator
dot should work "analogous to operator->" [as opposed to introducing
another special case set of rules] and identifi ed the two or three
small places in the ARM where statements would have to be changed to
accommodate operator dot. ["Add operator dot to the list of overload-
able operators on the top of page 330, ...]

The proposal didn't state much, because it wasn't asking for much.
What it did ask for was stated.

The only confusion with this proposal has been that many people haven't been able to make [or are unwilling to make] the analogy to operator->. Fortunately, you seem able and willing to make the analogy, and have gotten it! I agree entirely.

Here's a couple examples [that you already got] as to how to make the analogy:

Q: Does operator dot ever get called implicitly?

A: By analogy, does operator-> ever get called implicitly? Does any other operator get called implicitly? Answer: no. To be consistent with the overloading of the other operators, operator dot never gets called implicitly, it only gets called when someone explicitly puts a dot in their code, and the lhs is a object [or reference] of a class with operator dot overloaded.

Q: But what happens when someone does a smartref++ ???

A: By analogy, what happens when someone does a smartptr++ ??? Answer, in neither case is a '.' nor a '->' written in the code, so in neither case is the overloaded operator dot or operator-> called. Therefore smartptr++ must mean smartptr.operator++(int) and smartref++ must mean smartref.operator++(int)

[See ARM page 339 for people not yet familiar with distinguishing prefi x and postfi x operator++, operator--]

Q: But that means the programmer is going to have to write special code to handle operator++, operator[] etc.

A: By analogy to operator->, note that today for smart pointers programmers also have to overload a bunch of other operators if they want a sensible, complete, consistent set of operations.

Q: But overloading operator dot doesn't solve this, that, or the other problem!

A: By analogy to operator->: "But overloading operator-> doesn't solve this that or the other problem!" There are a lot of subtle problems in C++ that overloaded operators can't solve. These mainly relate to the "C" tradition equating pointers and arrays in un-type-safe ways. So be it, this is the "C" world that the C++ programmer is born into. We live in the world of our forefathers. So, I am not proposing by overloading operator dot to solve the entire C/C++ world's problems. Rather, I am simply proposing that the original reason for leaving out operator dot was not well-grounded, and therefore it is possible to remove a needless special case from the language, reducing its complexity, and improving its orthogonality.

operator dot useful....

Q: ....But look at this situation where a programmer could do bad things with operator dot....

A: But look at this situation where a programmer could do bad things with operator-> .... Look at this situation where a programmer can do bad things with operator<< .... Look at this situation where a programmer can do bad things with operatorˆ .... It is not the tradition of C nor C++ to legislate sensibilities to the programmer. Rather, it is the tradition of C and C++ to give the programmer plenty of rope by which they can either hang themselves, or by which they can scale high mountains....