Accredited Standards Committee*
X3, INFORMATION PROCESSING SYSTEMS

Doc No: X3J16/91-0121
WG21/N0054
Date: September 21, 1991
Project: Programming Language C+
Reply to: Andrew Koenig
AT&T Bell Laboratories
184 Liberty Corner Road
Warren NJ 07059 USA
`ark@europa.att.com`

Analysis of overloaded `operator.()`

Andrew Koenig
Bjarne Stroustrup

Background

Several years ago, C++ gained the possibility of implementing "smart pointers" by overloading `operator->()` for a class:

```
class T {
public:
        void foo();
        // ...
};

class Pointer {
public:
        T* operator->();
        // ...
};
```

The idea was that a `Pointer` object could then be used as if it were a `T*`:

```
void f(Pointer p) {
        p->foo();
}
```

Here, the expression `p->foo()` behaves like `p.operator->()->foo()` with the type constraints on `Pointer::operator->` implied by that behavior.

Analogously, several people have suggested that it should be possible to overload `operator.()` to allow the creation of objects that behave like "smart references."

* *Operating under the procedures of the American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 311 First Street NW, Suite 500, Washington DC 20001

On the surface, that idea is very appealing. Closer examination, however, reveals some problems. Once an ordinary reference is bound, no further operations are possible on it: any use of the name of the reference is immediately translated into a corresponding use of the object to which the reference is bound. This is likely to be unacceptable for a "smart reference," because what would be the point of having such a thing if its behavior were always identical to an ordinary reference?

This is not a problem for "smart pointers" because a pointer is an object distinct from the one to which it points. There is therefore no ambiguity between operations on the pointer itself and operations on its object. The lack of such a distinction for ordinary references makes it essential to create such a distinction for "smart references."

Proposal 1: distinguish by usage syntax

One possible solution to this problem was discussed in committee at the March 1991 X3J16 meeting: if an object `x` has `operator.` defined, then it will be used only for things of the form `x.y`. For example:

```
class T {
public:
        void foo();
        // ...
};

class Reference {
public:
        T& operator.();
        void foo();
        // ...
};

void f(Reference r, Reference* rp)
{
        r.foo();     // r.operator.().foo()
        rp->foo();   // rp->foo(); no call of operator.()
}
```

The call to `rp->foo` in `f` would call member `foo` of the `Reference` class, not of class `T`, because of the use of `->` instead of `.` in that statement. The idea is that `operator.()` would be used only when a `.` was explicitly used in an expression.

We discovered several problems with this approach. For example, sometimes member functions are called without using either form:

```
void g(Reference r, Reference* rp)
{
        r++;
        r.operator++();
        (*rp)++;
        rp->operator++();
}
```

Under Proposal 1, `r.operator++()` can only mean `r.operator.().operator++()`. What about `r++`? People expect these forms to be equivalent, so making it call `Reference::operator++` would surprise them.

But that is one case where `operator.` must be used without an explicit `.` appearing. Once we admit one case, it becomes much harder to argue against the others.

Proposal 2: distinguish through inheritance

Let's see what happens if we say that `operator.()` is called for *every* attempt to refer to a member of a "smart reference" class. There is now no trouble with our previous examples:

```
void f(Reference r, Reference* rp)
{
        r.foo();                // r.operator.().foo()
        rp->foo();              // rp->operator.().foo();
}

void g(Reference r, Reference* rp)
{
        r++;                    // r.operator.().operator++();
        r.operator++();         // r.operator.().operator++();
        (*rp)++;                // rp->operator.().operator++();
        rp->operator++();       // rp->operator.().operator++();
}
```

However, this raises a new problem. Suppose we want to define "smart references" that implement object semantics through use counts. In other words, we want `Reference` assignment to result in multiple `Reference` objects that refer to the same thing. To do this, we need to define `Reference::operator=` and `Reference::Reference (const Reference&)` that deal with the appropriate members of the `Reference` objects themselves.

Unfortunately, every attempt to use a member of the `Reference` class will immediately be forwarded via `operator.()`! In other words, under this proposal, a class with `operator.()` can have no other members because there's no way to get at them!

It doesn't seem that such classes would be much use. However, there is a trick that makes them useful:

```
class Refbase {
        void foo();
        // ...
};

class Reference: private Refbase {
public:
        T& operator.();
        // ...
};
```

The idea is to take all the stuff that a `Reference` needs to implement its particular semantics and put it into a base class. We then use `private` inheritance and derive `Reference` from that base class. When we want to get at a member of `Refbase`, we say something like this:

```
        ((Refbase*)(this)).foo();
```

In other words, we convert `this` into a base class pointer and get at the base class members that way.

Proposal 3: a simplification of Proposal 2

We believe that Proposal 2 is workable, but requiring users to define classes in pairs this way is clearly too complicated and error-prone. One possible simplification comes to mind: if a class with `operator.()` has additional members, allow those members to be used directly instead of forwarding their use through `operator.()`. For example:

```
class Reference {
public:
      T& operator.();
      void foo();
      // ...
};

void h(Reference r)
{
      r.foo();     // r.foo();
      r.bar();     // r.operator->().bar();
}
```

This would surely avoid many possible errors. Unfortunately, it does conceal a pitfall: if class T has its own bar member, users of the Reference class will unwittingly pick up Reference::bar when they meant T::bar. One might argue that it is the responsibility of the author of the Reference class to avoid such name clashes, but that fails to account for the possibility that class T might change later and the author of class T might not know of the existence of the Reference class.

A Broken Parallel

By applying operator.() even in cases where . isn't explicitly mentioned, we break the parallel with operator->(). Consider:

```
class X {
      // ...
      Y* operator->();
};

void f(X p)
{
      p->foo();    // p.operator()->foo()
      (*p).foo(); // no call of operator->()
}
```

Naturally, the parallel might be re-introduced by changing the semantics of operator->() so that operator->() is applied in both cases above. If that change were made, functions defined in the smart pointer class would be called in preference to functions defined in the referred class in the same way as functions defined in a smart reference class would be called in preference to functions defined in the referred class.

It isn't really useful to make that change, though, for smart pointer classes because the author of class X has the possibility of defining X::operator*() to provide appropriate behavior. This possibility does not exist for smart reference objects, though, because the reference somehow has to support all the operations defined for the object to which it refers.

Indeed, that is where the parallel really breaks down. The sharp distinction between pointer and object is blurred in the case of references, and that blurring is the cause of the trouble.

Broken Rules

Users can define operators for user defined types only. For example, a user cannot define a new meaning for operator+() of two ints. Furthermore, C++ does not define new composite operators based on user defined operators. For example, having operator+() and operator=() defined for a class X doesn't legalize

```
void f(X a, X b)
{
        a += b;
}
```

Proposal 3 for `operator.()` and its implication for `operator->()` violates both rules. This 'violation' can be explained by observing that the `p->m` to `(*p).m` transformation is somehow more fundamental than transformations such as `a+=b` to `a=a+b`. Note in particular that the first transformation is defined without weasel wording whereas the second has to take possible side efects of the evaluation of a into account.

Conclusion

At this point, we are convinced that more study is needed before a coherent proposal can be made for `operator.()`. We are writing down our thoughts so far in order to encourage others to contribute to that study. In particular, we need to look further for examples where a user-defined `operator.()` would be useful and consider the possible implications on the definition of `operator->()`.