
Document Title: Analysis of Overloadable Unary operator.()
Document Numbers: X3J16/92-0021 WG21/N0099
Author: Steven Kearns, Software Truth

This papers considers Jim Adcock's proposal (Doc X3J16/91-0140, WG21/N0073) for making operator .() overloadable. (Henceforth, this operator will be written as "operator dot()". It draws on the paper "Analysis of overloaded operator .()" , doc # x3j16/91-0121, by Andrew Koenig.

Summary of Proposal

Adcock suggests that operator dot() be overloadable in a way analogous to operator->(). Thus, if class Foo has an operator dot(), then "Foo f; f.bar()" is equivalent to "Foo f; (f.operator dot()).bar(). Just like operator->(), operator.() is only applied if an expression explicitly uses the dot operator.

Motivation for Proposal

An overloadable operator dot() would be useful for exactly the same reasons that an overloadable operator->() is useful. The most obvious is creating a "smart reference" class, say Ref<T>, that acts as much like a T as possible but can do something different in appropriate circumstances.

Koenig conclusively showed that it is infeasible to make a class that EXACTLY emulates a reference; the reason is that after creating a reference there is no way to operator on the reference itself, only on the object being referred to. Similarly, if a class Ref<T> always acted like a T, then there would be no way to act on the Ref<T>!

However, it is advantageous to make Ref<T> act as much like a T as possible. For example, I wrote a Val<T> class that converts any class T to have copy-on-write semantics; operator dot() would have useful in that endeavor because if T has a function Tfoo(), then you could say "Val<T> t; t.Tfoo(); " instead of, as is currently required, "Val<T> t; t.get().Tfoo(); ".

One could also argue on consistency grounds that if operator->() is overloadable, so should operator dot().

Feasibility of Proposal

Since the proposed behavior of operator dot() is exactly analogous to operator->(), which is a standard part of C++, there appears to be no problem in implementing operator dot().

Impact of Proposal on Code

As shown in the example of Val<T> given above, operator dot() can make code shorter and more understandable, in the same way that operator->() can make code easier to write and understand.

One can argue that it makes code harder to understand because operator dot() is another non-obvious operator being applied. However, this argument is weak because all operator overloading shares this property.

There is no cost of the proposal in terms of execution time or space or compile or link time. All existing programs operate the same with this proposal, because only a class with operator dot() defined is affected, and such an operator currently cannot be defined.

The proposal has no impact on static or dynamic checking of C++ code.

Things to be Careful Of?

Is it reasonable to require compiler vendors to implement operator dot()? Because the proposal for operator dot() is so similar to the existing behavior of operator->(), there appears to be no problem in adding such a feature in C++ compilers.

Is it hard to learn? It seems to me that the current state of affairs, in which operator dot() is an explicit exception, is harder to learn than making operator dot() behave analogously to operator->().

Does it lead to demands for other extensions? Possibly yes, since in the future some may perceive the need to overload operator.*(). However, I have never heard anyone suggest a need for this operator.

Arguments Against

One argument against the proposal is that it is possible to duplicate the behavior of operator dot() without this extension. For example, say we want to make a class FooRef which has an operator dot() that returns a Foo. Here is how it would look with operator dot():

```
struct Foo {
        int num1;
        int doit();
};

struct FooRef {
        Foo * fp;
        FooRef(Foo& fp0) { fp = &fp0; }
        Foo& operator .() { return *fp; }
};

main ()
{
        Foo foo1;
        FooRef fooref(foo1);

        fooref.num1 = fooref.doit();
}
```

Without operator dot() one could do something similar, by providing a function in FooRef for every function and member of Foo:

```
struct Foo {
        int num1;
        int doit();
};

struct FooRef {
        Foo * fp;
        FooRef(Foo& fp0) { fp = &fp0; }
        int& num1() { return fp->num1; }
        int doit() { return fp->doit(); }
};

main ()
{
        Foo foo1;
        FooRef fooref(foo1);

        fooref.num1() = fooref.doit();
}
```

Of course, there are some big disadvantages of this approach. If Foo ever changes, one must remember to make similar changes to FooRef. Also, this approach is impossible to generalize into a template; it must be done "by human hand" for each data type such as Foo.

doesn't address
usage, [illegible]
eg
a++
vs
a.operator++()

Necessary Changes

The changes necessary to implement this proposal are listed in Adcocks paper, and seem quite complete.

Conclusion

I believe that adding operator dot() to C++ as laid out in Adcock's proposal is the proper action. The language actually becomes simpler and more consistent with its addition, and I see no drawbacks to adding it.