Doc. no. P0700r0
Date: 2017-02-21
Programming Language C++
Audience: Evolution
Reply to: Bjarne Stroustrup (bs@ms.com)

Alternatives to operator dot

Bjarne Stroustrup

www.stroustrup.com

Abstract

This is a few comments on Hubert Tong and Faisal Vali's paper "Smart References through Delegation (2nd revision)" (P0352R0). To distinguish the two designs, I will refer to the inheritance-like/delegation/conversion suggestion as alt-dot (alternative to operator dot) and the current proposal as operator dot.

The C++ community asked for a way to overload operator dot to get "smart references" for decades and persistently. They did not ask for more implicit conversions; even the phrase "implicit conversion" triggers instant negative responses (reasonable or not) for many. People did ask for "delegation", but that was mostly in the early days where class hierarchies were all the rage. If the semantics of operator dot is best expressed as a conversion in the relevant sections of the standard, so be it, but don't mess with the interface for that – operator dot has a long and important history. By all means also improve the interface, but don't mess with the fundamental concept. In particular, smart references are not a form of inheritance because the referred-to type (the value type) is unrelated to the handle type.

I conjecture that if I had just implemented my favorite simple operator dot around 1990 (yes, I had one, see the references in the **operator.()** papers), we'd have been happy for decades and about 15 years ago, we'd have started to work on more advanced features as C++ evolved. Instead, we repeatedly tried to respond to people's demands for slightly different operator dot designs and each time failed to reach consensus. So, we got nothing. The time has come to focus on the operator dot design and overcome whatever technical problems and problems with WP phrasing remain, rather than again go on a long search for something perfect. Operator arrow isn't perfect either and it has served us well.

I couldn't be in Issaquah, sorry, and I didn't find the wiki summary very enlightning. Presumably partly because of that, I find the paper hard to read and it is woefully short of usage examples. Undoubtedly, I misunderstand something. I see a lot of very abbreviated criticisms of the operator dot proposal and find it hard to understand exactly what the alt-dot proposal offers to users (as opposed to standards writers), how it overcomes the perceived problems with the operator dot proposal for users, and whether it has complementary disadvantages.

First example

Consider first the first example in the alt-dot paper. The current design gives us this:

```
template<class X> class Ref {
        // …
        X& operator.();
        // …
};
```

Whereas the alt-dot design is identical except for

```
template<class X> class Ref: public using X {
        // …
        operator X&();
        // …
};
```

If the syntax is that similar, how about the semantics? Where are the benefits to the users?

Apart from the apparently redundant repetition of the type of the referred-to object in the alt-dot proposal, there appears to be no difference, so for this example, the proposals seem to be equivalent.

What is the purpose of the apparently redundant pseudo-inheritance/delegation notation? To enable the use of the conversion operator to be applied, I suppose, and to wiggle out of the complex name lookup rules that no one can remember.

Up to now, conversion operators have not been applied before a dot or when an operation is applied to an object:

```
struct B;

struct A {
        int a;
        B* p;
        operator B&() { return *p; }
};

struct B {
        int b;
        B& operator++() { return *this; }
};

int main()
{
        B b;
        A a {1,&b};
        a.b = 7;            // no
        B bb = a;
        ++a;                // no
        a.operator++();     // no
}
```

Should I think of the inheritance notation as simply an enabler of conversions before a dot and in expressions? In other words, am I right in assuming that this new kind of conversion operator would be applied in the examples above if a **public using B** declaration was in place? The built-in operator dot do not allow a conversion, but if a **public using B** declaration was in place without a conversion operator, would it be? If a conversion operator was present without the **public using B** declaration in place, I assume that the conversion would not be applied to a value before a dot or for **++a**.

I assume that the conjectured problems with understanding that **operator.()** is applied when operators are used (which I have never observed and don't expect to be real) would be match by a confusion about when a conversion operator is applied.

If we have a **Ref<X> r**, I assume that for alt-dot,

```
auto x = r;         // is x an X or a Ref<X>?

void f(Ref<X>);

f(r);               // does f receive a copy of r or a Ref<X> for a copy of r.operator X()?
```

the type of **x** will be **Ref<X>** just as for operator dot? And that **f()** will received a copy of the **Ref<X>**? This is a crucial point. This is the point where an earlier operator dot paper has inconsistent examples. If I am wrong in that assumption, all use cases need reexamining. The lack of usage examples in the alt-dot proposal (suggestion) makes it hard to comment.

I fail to see the "simplicity" of the conversion/inheritance-like/delegation approach as opposed to the **operator.()** approach that people have asked for in various forms for decades. What people want is smart references and the operator dot proposal delivers that. If some of the WP wording for **operator.()** is best expressed by recycling conversion rules in the standards text, so be it, but the key is that the feature provides a reasonably simple model of a handle that is a smart reference and resolves the key use cases as expected. I find it hard to read the WP text and 99.9% of the users cannot.

Random comments on alt-dot

For lack of time and of detailed examples of alt-dot use cases, I cannot do a point for point comparison, but here are a few observations.

Section 1 "Motivation"

++a invokes **operator.()** "might surprise". I will definitely surprise many if it did not and several key use cases would not work. Some of the early discussions were on exactly this point, but I consider the need clear and the rule easy to express comprehensibly. In the alt-dot proposal, it might be equivalently surprising that **++a** invoked the conversion operator.

Having to use **addressof()** to avoid recursive application of **operator.()** is definitely a pain, but the alternative considered was to introduce a new set of lookup rules for members of a handle class. If someone can solve this problem, I'd be quite happy.

The lack of equality between **p->foo()** and **(*p).foo()** is repeatedly mentioned as a weakness of the operator dot proposal, but note that the usual equivalences among operators (e.g. **[]**, *****, and **+**) are never guaranteed. If you want **.** and **->** to obey the usual equivalence, define both or neither.

The ability for **operator.()** to return an arbitrary type (and not just something that can have **.** applied to it) is a generalization. I would not describe it as an inconsistency. If we had uniform call syntax, it would be a very useful generalization (examples in the proposal), without uniform call syntax it is less so.

We cannot get access to member type names through **operator.()**. Of course not; **a.my_type x;** doesn't make sense in today's C++ either. And I don't think that alt-dot changes that or should change that. Similarly, there is no way of gaining access to a member type of the class of an object found by dynamic lookup (e.g. an object derived class accessed through a virtual function call).

Section 3 "Details and Technicalities"

Adapted classes simply are not "a form of inheritance" and that might be good, but it invalidates the argument that alt-dot is good because people are familiar with and like inheritance.

Adaption to a built-in type is not like C++ inheritance, but similar to an **operator.()** returning a built-in type.

Adaption through multiple adapters is not like C++ conversions, but similar to repeated use of **operator.()**.

In general, these details and technicalities do not outline a complete or systematic design, nor are the implications for users explored.

Section 8 "Pros and Cons"

Seems almost exclusively focused on implementation details and WP wording. The purpose of language features is to express ideas in user code.

Conclusion

The operator dot proposal is consistent with what people have asked for over the decades and handles the key use cases. The alt-dot proposal is not and I haven't seen the key use cases. From a user's perspective, I fail to see the simplicity of the alt-dot proposal.

If the alt-dot proposal changes the behavior of key use cases, we need to see and evaluate those use cases. If not, the issue is one of syntactic expression of ideas and how best to express the rules in the WP. The alt-dot proposal is heavy on language-technical points, but it is not a coherent proposal and essentially fails to address usage.

People have wanted **operator.()** for decades (for reasons documented in the current proposal). The main use cases are easily explained and understood. It is time to concentrate on that proposal and not impose another few years of delay and confusion about design direction.