Document number: **N2892=09-0082**
Date: 2009-06-09
Project: Programming Language C++
Reference: N2857=09-0047
Reply to: **Alan Talbot**
cpp@alantalbot.com

Some Concerns About Axioms

Abstract

I have some concerns about axioms as they are currently defined. These concerns can be expressed as two questions to which I can find no clear answers. In brief, my questions are: why would I want to write an axiom, and if I did, how would I get it right? Without answers to these questions, I feel that axioms as currently defined may lead to problems.

Question 1

If an axiom has no (guaranteed) effect on my code, why would I want to spend valuable time writing one?

Axioms do not explicitly control or (portably) affect the compiler or any other part of the build system. That is, I cannot use an axiom to reliably alter the behavior of my code. The relevant sections of the WD follow:

> 14.10.1.4 Axioms [concept.axiom]
>
> 4 Where axioms state the equality of two expressions, implementations are permitted to replace one expression with the other. …
>
> 7 Whether an implementation replaces any expression according to an axiom is implementation-defined. With the exception of such substitutions, the presence of an axiom shall have no effect on the observable behavior of the program. …

I think it is unlikely that programmers will be motivated to write axioms in the hope that the compiler might apply useful optimizations. In many cases I can apply the axiomatic substitutions myself and have a guaranteed, easily understood and visible effect. In situations where that is not possible or practical, writing axioms is at best an indirect and non-portable way to (possibly) optimize the code. Furthermore, axioms only affect constrained contexts, and most code is not in templates.

Of course, the same argument (more or less) can be made about inline. That also has no (portable) guaranteed effect. But there are two significant differences. First of all, inlining is well understood and the compiler has the information it needs to make very good decisions about what should and should not be inlined. (In fact, the use of explicit inline specifiers is probably not even necessary these days.) Second, the cost to the programmer is very small (one keyword per function) and the likelihood (and the cost) of getting it wrong is very low. Inline is also available for all code, not just templates.

Question 2

If I write an axiom, how do I know whether it is correct?

Although axioms are not guaranteed to be used, from section 14.10.1.4 we know that an axiom *could* have a profound effect, so if I'm going to write one, I'd better get it right. There is no guidance in the current definition of axioms that would help me do so. While this is true of many other facets of the language, there is a major difference in the case of axioms: in all other cases I can find out whether I have written something correctly by observing the results. I have the compiler (at several levels), the linker, runtime errors, and finally the observable program behavior to tell me if something is wrong.

With axioms, the only help I have is a check of syntactic correctness by the compiler; once the syntax checks, I'm on my own. But syntax is the easiest part to get right. The only empirical way to tell if the semantics of axioms are correct is by subtle and hard to measure optimizations, which of course may not happen. Much more worrisome is that the substitutions could just as easily be pessimizations if I get the axiom wrong.

For example, here is an excerpt from the WD:

23.2.6.2 Member container concepts [container.concepts.member]

```
auto concept MemberContainer<typename C> {
    ...
    size_type C::size() const {
        return distance(this->begin(), this->end());
    }
    ...
    axiom MemberContainerSize(C c) {
        (c.begin() == c.end()) == c.empty();
        (c.begin() != c.end()) == (c.size() > 0);
    }
}
```

Now what happens if I write something like this? (I've ignored the Container-MemberContainer mapping for clarity.)

```
template<MemberContainer C> void process_front(const C& c)
{
    if (c.begin() != c.end())
    {
        ... // do something with c.front()
    }
}

void my_algorithm()
{
    forward_list<...> fl;
    ... // put something into fl
    while (fl.begin() != fl.end())
    {
        process_front(fl);
        fl.pop_front();
    }
}
```

This seems to me to be pretty reasonable code. I have a process_front routine that does something with the first element of any container, and an algorithm that uses a forward_list in some way and calls process_front from a loop. The idioms I use here are common STL container idioms. So what is the problem?

The MemberContainer concept is auto, and so a map will be generated for forward_list. But forward_list was deliberately designed without the size() member so users wouldn't write code that assumed that size was O(1)[1]. This means that a size() member will be provided by MemberContainer using the default O(n) implementation in terms of distance(). Because of the MemberContainerSize axiom, the compiler is then free to substitute `begin() != end()` with `size() > 0` in process_front. So what looks like an O(1) test for not-empty becomes an O(n) size calculation.

In my opinion this problem is very hard to see. It's unlikely I would think of it just looking at this code unless I stepped in with the debugger and found myself in distance(). Worse, it would work fine on a compiler that either didn't do axiom substitution or chose to substitute in the other direction. Even worse, the substitution probably won't happen in debug mode, where optimizations are typically turned off. I don't know whether a compiler could figure out that this substitution was a bad idea, but such intelligence would likely be in the realm of unspecified behavior (since compiler vendors don't like to guarantee specific optimizations).

The preceding example only produces a faulty optimization, not incorrect behavior. But consider this axiom (also from the MemberContainer concept in 23.2.6.2):

```
axiom MemberFrontInsertion(C c, value_type x) {
    x == (c.push_front(x), c.front());
}
```

The implications of this substitution are sobering. A compiler could easily decide that `x` is much more efficient than `(c.push_front(x), c.front())`, and this axiom says that they are equivalent. The push_front, front expression is a reasonable thing for someone to write, and if it gets quietly replaced by `x`, the behavior of the code that contains it will be superficially similar, so the missing insertion will be very hard to find.

My point here is that writing axioms is subtle stuff indeed. Perhaps whoever wrote the MemberContainerSize axiom above knew that a compiler would not, or could not, pessimize in the way I've illustrated, but I don't think it's reasonable to ask that level of expertise of our users. And as MemberFrontInsertion shows, there are axioms in the current WD that have been identified as potentially incorrect (by people more expert than myself). If members of the Committee can't get them right, or can't agree as to what is right, how can we expect average users to do so?

[1] The whole point of forward_list is to provide a minimal forward-linked list that has no overhead over the naïve C implementation, so a size calculation would have to be O(n).

Conclusions

Without a convincing answer to both these questions, I believe we should either remove axioms or change their definition. As currently defined, they are sort of a syntax-checked comment, but unlike a comment they can have visible and potentially dramatic behavior. I believe this behavior is far more of a liability than a benefit.

The promise of third-party tools that do useful things with axioms is interesting and compelling, but such tools may or may not be available or useful to any particular programmer. Furthermore, these tools do not yet exist, so I question whether we can know today what information they will need.

Tools that use axioms do not depend on the substitution behavior of compilers. If axioms are, in effect, comments, then perhaps they should actually be comments. A standard syntax for expressing axiomatic relationships in comments would allow third-party tools to operate without causing the problems I discuss above, and it would be easy to include version information to facilitate backward compatibility if there is a need to change the syntax in the future.

Another option would be to prohibit compilers from performing axiom substitutions. I believe this functionality could be added later without breaking things once we have a better idea of how axioms should work. Without the substitution language, the answer to the above questions is easy: 1 – you wouldn't unless you were using an analysis tool that made use of them, and 2 – it doesn't matter to your code but you can check them with your analysis tool.

One concern that was raised about removing axioms went roughly as follows (I hope I have not distorted the meaning here): since axioms are part of the formal specification of the language, we cannot change them once they are in the library, nor can we add them later since that would change existing code, therefore we had better get them right first time. I believe that this is not quite correct. Since (correct) axioms by definition do not change code behavior, changing them (or adding them later) cannot affect existing code so long as they were correct to begin with and so long as the changes or additions are correct.

This leads me back to the same conclusion. As currently defined, axioms must be correct to ensure they have no visible effect on code. We don't yet seem to be sure how to get them right, and we don’t have any implementations to help us, so for now we should either remove them or change the definition.

Acknowledgements

Thanks to Jens Maurer for providing lots of information and ideas, and for reviewing a draft. Thanks to Doug Gregor for reviewing a draft and making helpful suggestions. Thanks to Alisdair Meredith for pointing out the problem with MemberFrontInsertion.