X3J16 / 93-0138
WG21 / N0345
09/27/93
John Bruns
NationsBanc-CRT

Lifetime of temporaries explicitly bound to references

Most references are bound as function arguments or function returns. Assuming they are bound to temporaries, the lifetime of the temporary is now defined as the life of the statement. This works well since the lifetime of such references matches. A question arises whenever the reference is explicitly bound. In Munich, we resolved that the temporary is bound to the life of the reference. The question is what temporary. Even where that is specified in the paper, did we make the correct decision.

```
Class B {...};                     // base class
Class D : public B {...};               // derived class
Class C {public: D d; ...};        // containing class
extern C f();

C c1 = f();                   // #1  copy class object
const C & cr = f();           // #2  bind  cr to temp result of f()
```

Here there is no dispute, the temp bound to cr is the result of the function f(). The difference between the two lines is that in case #1 a copy must be made. Case #2 binds in place.

```
B b1 = D();                   // #3  copy the B from temp  D()
const B &br1 = D();           // #4  bind to the B base in D()
```

It gets slightly more complicated when we have a derived class. Yet even here there is no controversy. Everyone (I think) agrees that br1 in case #4 is bound to a temporary D object created by the D() constructor. The result of any overridden virtual functions using br1 will be calls to the appropriate D:: member functions. In case #3, it is evident that a new B b1 is constructed, all member functions called using b1 will be B:: members. The D object has been truncated to a B.

```
B b2 = f().d;            // #5  copy the B from  the f()'s temp's d.
const B & br2 = f().d;   // #6  WHAT does br2 bind to.
```

In case #5, similar to case #3, a copy of a B is made by truncation. Now comes the controversy. What does br2 bind to? What is the lifetime of the temporary C object created by f()??? There are a number of possible options.

The logical thing to do is to bind br2 to the B base of the D member object (d) in the temporary C.

That is certainly what would happen if the expression were used as the argument to another function

```
     extern  g( const B &);
     g(f().d);                    // #7
```

In this case, I would be somewhat disturbed to find anything else happening.   Since the lifetime of all temporaries in this expression is the statement, there will be no problem. Since there are three (or two depending on how you count) temporaries in statement #6, what are there lifetimes. Specifically there is the base B to which the refernce is bound, the D object C::d, and finally the C object which contains them. Our lifetime document specifies that the B remains. Case #4 suggests that the D remains and that virtual functions called on br2 call D:: member functions. However, nothing is said about the C. Yet, by definition, we can't destroy the C without destroying the B also.

One suggestion was to make a copy (I would assume of the D object) and bind that. There are some problems with that. The first one is that the copy constructor for D might not be accessible. If the class designer doesn't want D's (or B's) copied, the compiler should not break this encapsulation. The second one is that D might have knowledge of its enclosing object. While this might not be the best programing practice it is certainly done. In fact is can usually be done safely since objects are alway (up to now) created and destroyed as a unit.

Consider the D constructor takes a reference to some object of type T that is inside C. The class designer never uses D except inside classes that also contain T's. He thinks its safe, since when C is built, and the D part is built, he always has the T. It can't go away until the C is destroyed. This will work the way he expects in #7. It works the way he expects now with long lifetime compilers for case #6. Why should it break now?

To make #7 work as #6, the simple solution is that explicitly bound references must keep all enclosing temporaries around until the reference goes out of scope.

Other possibilities are:

Make case #6 illegal. I have a lot of sympathy for this solution since it can always be coded around, often with safer results. I suspect people actually using such constructs will have less positive opinions.

Copy the B temporary and bind to it. This is unlikly since it give very different results (**and quietly**) when used with a named variable vs a temporary.

Copy the D termporary and bind to it. This won't fail in as many circumstances, but it will in the case sighted above.

Because of the quietly changing nature of the code in either of the copy sceanarios, I don't think we can accept these solutions.

Another similar issue

Consider a string class with a substring helper class. The substring class is private with string as a friend class. The string class uses substring to generate a pointer into himself, along with a data range. Since the class has no public constructors, the user can't capture a substring except with a reference. As we have already seen, the statement lifetime of temporaries protects us whenever this construct is used in a single statement.

```
string s, t;
cout << (s + t).substr(1,5);                // works fine

const substring & sr = (s + t).substr(1,5);
cout << sr;                                 // core dump ??
```

The lifetime paper explicitly stated that the temporary sr was bound to was the result of the substr() member function, not the (s + t) operator. I think this was a bad choice. A programmer knowing nothing of the internals would expect this to work in the second case as well as the first. Techniques of this sort are useful and widely used. The class designer can protect against temporary expressions causing problems in all but the explicit binding case.

Possible solutions are:

Ignore the problem, this is what we agreed to in Munich. Class designers would simply document that this will cause problems and hope everyone reads the warnings. I don't think the users of the language will thank us.

Eliminate the ability to explicitly bind to a temporary. This would be ok with me, since I see this construct as fraught with peril . Unfortunately, it is a part of the language, and I doubt that removing it now is an option.

Extend the lifetime of the temporaries involved. I recommend we revise the lifetime of temporaries rule to state:

For any statement explicitly binding a reference to a temporary, the lifetime of all temporaries in the statement are extended to match the lifetime of the reference.

While this will not cure all possible dangling refernce problems, it would make the language consistent. Everything that works in statement scope would also work when explicitly bound as a reference. It also fixed the problem of containing objects consistently. As always, the as if rule applies. That is a compiler can destroy any temporaries early if it can guarantee that there is no possible side effect.