Accredited Standards Committee[1]
X3, INFORMATION PROCESSING SYSTEMS

Document Numbers: X3J16/93-0140
WG21/N0347
Date: June 18, 2003
Reply To: Bill Gibbons
bgibbons@taligent.com

Local Disambiguation of Dynamic Cross-Casts

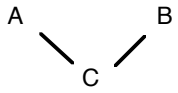
Introduction

A dynamic cast [1],[2] may fail because the target class appears more than once in the object's complete class hierarchy. An explicit exception is made for strict downcasts, which are allowed even if the target class is duplicated elsewhere in the hierarchy. So the error only occurs with cross-casts.

This rule is overly restrictive. In particular, it may cause an existing class to stop working when it is included in a larger hierarchy. When the class is part of a vendor-supplied library, it may be impossible to change the code to fix what is arguably a problem which should not have existed in the first place.

Example

Given the simple class hierarchy: (derived classes at the bottom; everything public)



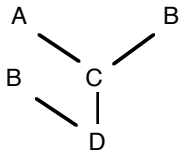
And the code:

```
void f(C *c) {
    A *a = c;
    B *b = dynamic_cast<B*>(a);
}
```

There is no reason for the dynamic cast ever to fail. The target class is sitting right there, in plain sight. Under the current working paper, it should always work. Right?

Wrong.

The author of this hierarchy has no way of knowing whether it will be included in a larger hierarchy. If, for example, it gets used in the following way:



The dynamic cast will fail, *even though the original hierarchy is intact*. This may come as a complete surprise to the author of the class. It may come as an even bigger surprise to the clients of the class, who may have purchased a class library and have no access to the source code.

[1] *Operating under the procedures of the American National Standards Institute (ANSI)*

The existing rule makes it difficult if not impossible to use a cross-cast in a hierarchy which may be used as part of a larger hierarchy. This is a serious impediment to the reuse of code.

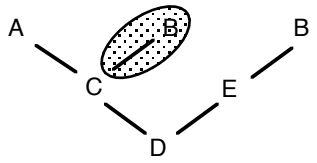
Local Disambiguation

One solution to this problem is to take advantage of local information in the hierarchy to select a base class which is locally unambiguous. If some part of the hierarchy contains both the starting class of the cast and a single instance of the target class, the cast is unambiguous within that portion of the hierarchy and its behavior should not change just because the subhierarchy has been embedded in a larger hierarchy.

The exact rules which would control local disambiguation are given later.

Distracters

The counter-argument is that a locally disambiguated cross-cast may be changed quietly by the addition of a "distracter" class in the hierarchy. For example:



Suppose the original hierarchy did not contain the shaded base class. The following code:

```
void f() {
    A *a = new D;
    B *b = dynamic_cast<B*>(a);
}
```

would find the single instance of `B`, namely the rightmost one in the above graph. Suppose the declaration of `C` were changed to add `B` as another base class, as in the shaded area. The new base class would be a distracter; it would silently change the result of the dynamic cast.

What effect would a distracter have on the design of a hierarchy? There are two cases:

1) The author of `A` did not know about the first `B`. Then presumably the author of `A` was not picky about which `B` the cast found; any `B` would serve his needs. The distracter does not cause any problems.

2) The author of `A` knew about the first `B`. Then A is designed to be used as part of a `D` hierarchy. The author of `D` should be aware of this, and should know that adding a second `B` might change the behavior of `A`. So the distracter would not be added by accident, and would not cause a surprising change.

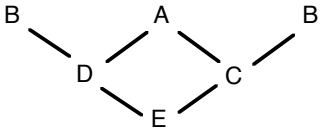
Either way, the distracter should not cause a surprising change to the behavior of the program.

Casts To Virtual Base Classes

What if the target class is a virtual base class? Does this have any effect on local disambiguation? No. The intent is to find a subgraph of the hierarchy in which the dynamic cast is unambiguous. If such a subgraph exists, the cast is locally unambiguous. If not, the cast is truly ambiguous and must fail. This holds regardless of whether the target class is a virtual base class.

Casts From Virtual Base Classes

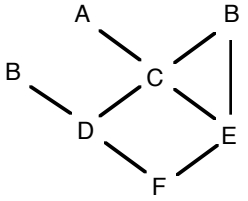
On the other hand, if the starting point of the cast is a virtual base class, the notion of locality is more difficult to apply. For example:



Now the cast from `A*` to `B*` really must be ambiguous, even though there exists a subgraph in which it is unambiguous. Actually there are two subgraphs in which it is unambiguous, and they yield different results.

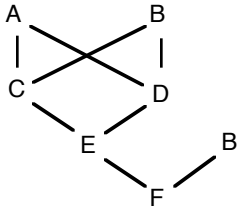
Notice that this does in fact violate our "can't change the behavior by deriving from it" rule, since the behavior of the `C` hierarchy was changed be embedding it in the larger hierarchy. Why? Because in some sense the `C` hierarchy is "open", since it has a base class which it may have to share with some larger hierarchy because the base is virtual.

But not all such casts from virtual bases are problematic. Some of them are well defined:



Here a cast from `A*` to `B*` is well defined even though both are virtual base classes. The difference is that here there exists a unique local context for the unambiguous cast which is "more local" than any subgraph in which the cast would be ambiguous.

Even when the "most local" context is not unique, the cast may be well-defined:



Here both the `C` and `D` subhierarchies are local contexts where a cast from `A*` to `B*` is unambiguous. But since the result is the same regardless of which is used, the cast is well-defined.

Proposal

The relevant wording is in §5.2.6/4; it refers to the expression "`dynamic_cast<T>(v)`", although it really means "`dynamic_cast<T*>(v)`" or "`dynamic_cast<T&>(v)`". (The wording has been slightly changed to correct an editorial error, and the cases have been numbered; otherwise the quote is verbatim.[2])

1) If the complete object pointed (referred) to by `v` is of type `T`, the result is `v`.
2) Otherwise, if in the complete object, `v` points (refers) to an unambiguous base class sub-object of a `T` object, the result is a pointer (reference) to that `T` object.
3) Otherwise, if the type of the complete object has an unambiguous public base class of type `T`, the result is a pointer (reference) to the `T` sub-object of the complete object.
4) Otherwise, the run-time check *fails*.

I propose replacing case 3 with the following words, subject as always to editorial changes.

3) Otherwise, if the type of the complete object has a base class of type `T`, consider all derivation paths from the unique base class within the complete object's hierarchy corresponding to the subobject to which `v` points (refers) to the complete object's class. If, on each such path, the first (least derived) class which has `T` as a base class has `T` as the same unambiguous public base class, the result is a pointer (reference) to that `T` sub-object of the complete object .

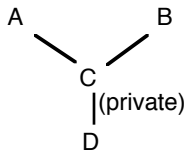
The revised rule has two important consequences:

Some cases which previously failed because they were considered ambiguous now succeed due to local disambiguation. These cases are described in the examples above.

The access rules for cross-casts apply only to the relevant local part of the hierarchy. This is in keeping with local disambiguation, but it also affects unambiguous cases as described below.

Cross-cast Access Checks

One surprising effect of the existing rules for cross-casting is that existing code may break when a class is embedded in a larger hierarchy even when no duplicate base classes are involved. Consider the hierarchy:



If the complete object is of type `C`, a cross-cast from `A*` to `B*` succeeds. But if the complete object is of type `D`, the cross-cast fails because `B` is not a public base class of `D`. Surely `D`'s access to `B` should have no effect on the `C` sub-hierarchy.

The proposed rules for local disambiguation make this cast succeed because the access depends on `C`, not `D`.

One variation on the access check would be to allow the cross-cast if any path had public access, rather

[2] The working paper does not appear to allow a dynamic cast to the exact complete object type rather than one of its base classes. This is surely an oversight.

than only if all paths have public access. This would change rule (3) to:

> 3) Otherwise, if the type of the complete object has a base class of type `T`, consider all derivation paths from the unique base class within the complete object's hierarchy corresponding to the subobject to which `v` points (refers) to the complete object's class. If, on each such path, the first (least derived) class which has `T` as a base class has `T` as *the same unambiguous base class, which is a public base class on at least one such path*, the result is a pointer (reference) to that `T` sub-object of the complete object .

where the change is in italics. The reasoning is that if there exists a class through which the cast could be done in two parts, as a downcast and an implicit upcast, then the cross-cast should not be disallowed for lack of access.

Conclusions

The current dynamic cast rules in the working paper make cross-casts nearly useless in class libraries, since any reuse of a hierarchy in which cross-casts are used may cause those casts to fail. It is possible to relax the restriction in a way which permits class libraries to use cross-casts safely. Although this change also makes it possible to change the behavior of a cast by adding a distracter base class, there is little real danger of a surprising behavior change.

The improved opportunity for code reuse and for use of dynamic casts in class libraries justifies changing dynamic casts to permit local disambiguation and local access checking.

References

[1] Stroustrup & Lenkov. Run-Time Type Identification for C++. *X3J16/92-0121 & WG21/N0198.*

[2] Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++. *X3J16/93-0062 & WG21/N0269, Section 5.2.6.*