X3J16/90-0049

A Proposal to Revise C++ Subtyping Rules

Alan Snyder

2 July 1990 (DRAFT)

17-May-90/23-May-90/29-May-90/1-Jun-90/13-Jun-90/25-Jun-90/26-Jun-90/28-Jun-90/2-Jul-90

Introduction

This note discusses a proposed extension to the C++ subtyping rules. In C++, the type "pointer to a derived class" is a subtype of "pointer to a public base class". Similarly, the type "reference to a derived class" is a subtype of "reference to a public base class". (To simplify the presentation, we use only pointer types henceforth; reference types are handled similarly.) The proposed extension relaxes a restriction on a derived class declaration in a way that is consistent with the standard notions of subtyping in object-oriented languages.

Overview

In the "standard" view of subtyping in object-oriented languages, a subtype is permitted to revise the interface specification of an inherited method by generalizing the types of the arguments and/or restricting the types of the result. Such revisions are type-safe.

The analogy to a method in C++ is a virtual member function; the analogy to subtyping is public class derivation. C++ does not allow either kind of revision for virtual member functions of publicly derived classes.

Although C++ permits a derived class to "redeclare" a virtual member function with generalized argument types, the derived class function does *not* override the base class function; it cannot be invoked from the base class context. Instead, the new function is viewed as a distinct instance of an *overloaded* function, almost as if it had a different name than the base class function.

A programmer can "work around" this limitation of C++ by defining *two* member functions in the derived class. One function has the same argument types as the base class function. The other function is defined with the generalized argument types. The function with the restricted argument types is defined to invoke the function with the generalized argument types. This solution achieves the effect of generalizing the argument types of a method in a subtype, at the cost of additional housekeeping by the programmer and the execution cost of the additional function call.

Restriction of the return type of a virtual member function in a derived class is *illegal* in C++. This rule reflects the fact that C++ overloaded functions are resolved based on argument types, not return types. In this note, we describe a proposal to extend C++. The proposal allows a *limited* form of overloading based on return types. This limited overloading can be used in a way similar to that described in the previous paragraph to provide the effect of restricting the return type of a method in a subtype.

An example

The proposed extension to C++ is illustrated by the following example:

```
class BB { ... };
class DD : public XX, public BB { ... };
class B {
     int b1, b2, b3;
     public:
     virtual BB* f() = 0;
     };
class D : public B {
     int d1, d2, d3;
     public:
     /* virtual */ DD* f();
     };

DD* D::f () {
    dd* p = foo ();
    return p;
    }
```

The behavior we desire is as follows. We want the function *D::f* to be invoked when the member function *f* is invoked on an object of class *D* in a context where the object is known as an object of class *B*. In this situation, we want the return value of type *DD** to be converted to the expected type *BB**.

C++ currently considers the declaration of class D to be in error. In the current definition of C++, a derived class function cannot differ from a base class virtual member function only in the return type. The proposal relaxes this rule to permit the above example.

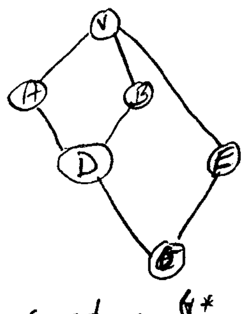
The proposal

If it is to be accepted, it should be orthogonal

The proposal is as follows.

We retain the rule that a class cannot *declare* two or more member functions with the same name and argument types but different return types. However, we extend C++ to allow a derived class to declare a member function that differs from a *virtual* base class member function only in the return type, just as it now can for a *non-virtual* member function. (Note that if a class is defined in current C++ with multiple base classes, there can be more than one base class function with the same name, same argument types, but different return types.)

Weird Restriction The virtual base scenario is the problem



We define the new member function to *override* the old member function(s). That is, an invocation of the member function in a base class context on an object of the derived class will invoke the new member function. The return value will be converted to the type specified by the member function declaration in the base class.

For the derived class to be legal, an implicit conversion must be defined from the new return type to each old return type.

As in current C++, the old member functions will be *hidden* in the derived class. Thus, a class can contain at most *one* visible member function with a given name

G returns G*
D returns D*
V returns V*
all are normalized to V*
now
if pD points to a G*
where is V*? If V* is known, where is D*
if PE points to a G*
where is V*? If V* is known, where is E*
> where is G*?

and argument types. As in current C++, the derived class must either define the function or declare it pure.

In the above example, invoking f on an object of class D in the context of class B will invoke D::f, and the return value will be converted from type DD* to type BB*.

Implementation

The main implementation issue to be addressed is how to cause the conversion from the new return type (DD*) to the original return type (BB*) when the virtual member function (f) is invoked on a derived class (D) object from the context of the base class (B), but not when it is invoked from the context of the derived class (D).†

The implementation is fairly straightforward, given the right perspective. The basic idea is as follows. Currently, when a virtual member function is overridden, the C++ compiler reuses the same vtable entry for the new member function. In the case where the return type is different than the return types of all of the base class member functions, the compiler will allocate a *new* vtable entry for the new member function. (If the return type is the same as the return type of one of the base class functions, then the vtable entry for that function can be reused.) For each of the old vtable entries (except the one that is reused, if any), the compiler creates a function definition. The function will invoke the new virtual member function on *this* and perform the appropriate conversion of the return type.

BP B::f()
vtY
TH D::f()
vtY

In the example, when the compiler sees the declaration of f in D, it assigns it a *new* vtable entry. Thus, there are *two* vtable entries. One is defined for class B; we will label this vtable entry B#f. The other is defined for class D; we will label this vtable entry D#f.

-+adj: *(*this + y))()
What to do with the
return address. Reentrancy
problems : is a full frame
alternative
what about ...
call
functions.
Yow.

The compiler also defines two function symbols: for explanatory purposes, we will call these functions D::B::f and D::f. The compiler will create a function definition for D::B::f that invokes f on *this* and converts the return value from type DD* to type BB*; the address of this function will be stored in the B#f vtable entry for class D. The address of function D::f will be stored in the D#f vtable entry for class D; no actual code for this function will be generated until a function definition for D::f is encountered.

The definition of D::B::f is effectively:

Must be a virtual call.

```
BB* D::B::f () {
    dd *p = f (); // call the virtual member function
    return p;     // converts DD* to BB*
    }
```

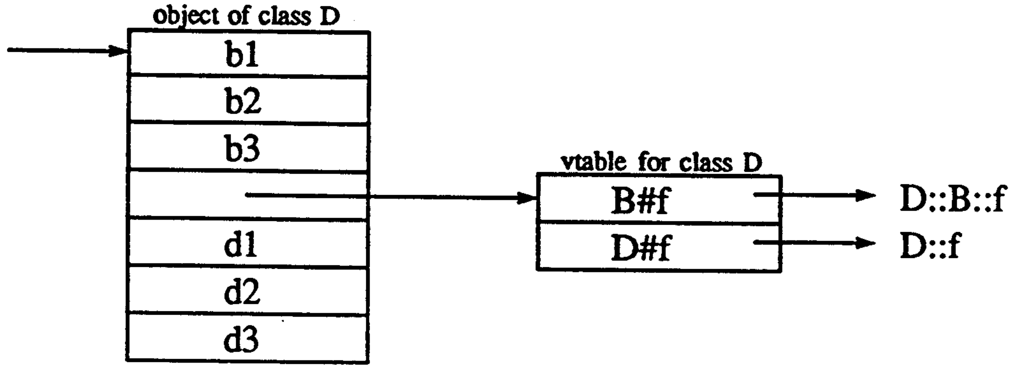
What about ...

[†] Note that this conversion may require actual computation in some cases, depending upon the implementation of C++. The example above requires conversion in the AT&T 2.0 C++ translator implementation because of multiple inheritance. In effect, an instance of class DD contains a "subobject" of class DD and a "subobject" of class BB. A BB* must point to a "subobject" of class BB. A DD* must point to a "subobject" of class DD. When a DD* is converted to BB*, the pointer to the DD "subobject" is replaced by a pointer to the associated BB "subobject". These pointers are different.

A compiler will often be able to generate better code for this function as a special case than it would by simply compiling the above definition.

The compilation of a definition for D::f is handled as normal. The function definition for D::f will be accessed via the D#f vtable entry for class D.

The following diagram illustrates the implementation of an object of class D as defined by the above example:



Note that D::f, the function defined by the programmer, is invoked whenever the f member function is invoked on the object in a D context. The compiler-supplied function D::B::f is invoked whenever the f member function is invoked on the object in a B context. The compiler-supplied function D::B::f invokes the programmer-supplied function D::f via the D#f vtable entry and then converts the return value to the expected type (BB*). Alternatively, D::B::f could call D::f directly. In this case, each subsequent overriding of f in a derived class of D would require a new corresponding compiler-supplied function for the B#f vtable entry.

Per level.
what about multi-level cascaded calls ?

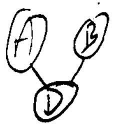
The implementation cost of this solution is low: one extra compiled function and one extra vtable entry. The indirection can be optimized, or eliminated by compiling two copies of the programmer-supplied function, one with and one without the conversion of the return value. In the (likely) case where no actual conversion is required between the two return types, then the two vtable entries can be collapsed back into one, and only one function need be compiled.

An analogy

Not the
this adjustmnt
is pre-call
adjustment. this
is post-call
adjustment.
a Thunk won't
work.

The implementation issue we have discussed is the pointer conversion required upon return when a member function definition is shared (because it is virtual) by a base class and a derived class that declare different return types for the same function. Upon reflection, one can see that this conversion is analogous to the pointer conversion required on the implicit *this* argument when a virtual member function definition is shared by a base class and a derived class. Not surprisingly, the same implementation technique can be used for the *this* conversion. This technique is described by Michael Ball in the June 1990 issue of the C++ report, and in §10.8c of the Annotated C++ Reference Manual. Using this technique, the derived class vtable points to a different function (called a thunk) which converts the *this* pointer and then calls ("jumps" to) the base class function.

what about:



A* A::f();
B* B::f();
D* D::f();

Pointers to members

How does the above proposal affect pointers to members? The old and new member functions are different members, and have different types. They can be individually named by pointers of different types.†

However, it is possible to extend the definition of pointers to members to allow conversion from a pointer to the old member function to a pointer to the new member function. Suppose a base class *B* declares three virtual member functions *f1*, *f2*, and *f3* of type *FT=function returning T*, and suppose a derived class D redefines *f1* and *f2* to be of type *FU=function returning U*, where an implicit conversion is defined from *U* to *T*. We propose to allow conversion between *pointer to FT member of B* and *pointer to FU member of D*. This conversion makes sense only if the actual value is *f1* or *f2*. The implementation would have to convert between two corresponding pairs of values, one being the *B::f1, D::f1 pair*, the other the *B::f2, D::f2* pair.

(Would require 3 fields: pre-adjustor, post-adjustor, function addr.

A variation

We have considered a slight variation of the above proposal. In this variation, we allow the programmer to provide the definitions that override the old member functions, instead of the compiler providing implicit definitions.

The only issue is how the programmer should *name* the old member functions when providing the new definitions. One possibility is to introduce a compound name syntax, such as the D::B::f notation used above. (This notation refers to the definition of f in D that overrides the definition of f in B.)

This variation is awful. Whatever about re- Z* → D* relationship: is no analogue to built in: in any scope, only one name arg set may exist.

Consider the following example:

```
class X { public: virtual int f (); };
class Y { public: virtual char f (); };
class Z : public X, public Y {
    public:
      float f (); // new function
      int f ();   // overrides X::f
      char f ();  // overrides Y::f
      };
float Z::f () { ... }
int Z::X::f () { ... }
char Z::Y::f () { ... }
```

No way! Impossible to use. No way of disambiguating use of those functions, in context of Z.

Class Z contains five member functions named f. Their full names are Z::f, Z::X::f, Z::Y::f, X::f, and Y::f. Function Z::f is invoked when f is used unqualified

†In the 2.0 cfront, when a virtual member function is overridden in a derived class, it is impossible to point to the old and the new member functions individually. This behavior is the result of using the location of the vtable entry as the value of the pointer. The vtable entry is shared by the two member functions in objects of the derived class. This behavior is inconsistent with normal member function invocation, where one can invoke either f (the vtable entry), B::f, or D::f (the specific function definitions).

in the Z context. Function Z::X::f overrides X::f; it is invoked when f is invoked on a Z in an X context. Function Z::Y::f overrides Y::f; it is invoked when f is invoked on a Z in a Y context. There are three vtable entries for class Z.

Under this variation, to get the desired behavior in the original example, the programmer would explicitly define the function D::B::f above. The issue of compiling such a function efficiently would have to be addressed by an implementation.

Analysis

Maybe

The proposal achieves the effect of refinement of virtual member function return types that are pointers to class instances, which is consistent with the standard subtyping rules for object-oriented programming languages. In this case, it is not unreasonable for the programmer to view the base class and the derived class as providing two definitions for the same "operation", just as with ordinary overriding of virtual member functions.

However, the proposal is not limited to return types that are pointers to class instances. It allows any refinement such that there is an implicit conversion from the new return type to the original return type. All the C++ compiler needs to know is how to convert from the actual return type produced by the derived class function to the expected return type defined by the base class. Thus, for example, the base class could define a function to return a float, and the derived class could redefine the function to return an int.

What about NV in ABC? & temporaries?

In this way, the proposal goes ~~beyond the notions of~~ conventional object-oriented programming. In effect, it allows a virtual member function to return *semantically different values* depending upon the context of the invocation.† Allowing the programmer to specify arbitrarily different code for D::f and D::B::f (the "variation") is even more alien to object-oriented programming.

into unreadable

Acknowledgments

This proposal has benefited greatly from discussions with Peter Canning and Dmitry Lenkov. Dmitry suggested the implementation described in this note.

I am not surprised.

What about function args? A::f(A*); B::f(B*); ?

†Even pointers to objects of class type can produce different return values, in the case of multiple inheritance. However, these values are different only in an implementation sense: they are two different pointers to the *same object*. One could argue that the programmer should not see these as being different values, but merely different implementations of the same value. It is unfortunate that C++ pointer equality allows the difference in values to be visible to the programmer, although the problem is probably rare in actual practice.