X3J16/91-0132
WG21/N0065

What Ever Happened to Generalized Overriding?

[Editor's prologue] When multiple inheritance is used to create a new class, there is a possibility that there will be collisions, that is, names used in both of the original classes, but not having the same meaning. In the case of virtual function name collisions handling, this problem proved to be difficult. At the July 1990 meeting, Bjarne Stroustrup brought before the committee a proposal for the generalized overriding extension (X3J16/90-0098) which addressed this problem.

Following are two articles on generalized overriding, the first was written by William M. Miller shortly after the July meeting and the second written by Andrew Koenig after the issue was finally resolved.

William M. Miller

The original problem to which this proposal was addressed occurs when a class is multiply derived from two or more base classes, each of which defines a virtual function with the same name and argument signature. The existing C++ definition accepts this situation and provides a default which is the correct resolution in most cases: the derived class can provide a definition which overrides the corresponding virtual functions from all base classes. That this is not appropriate in some cases can be seen in the following example, created by Bjarne Stroustrup:

```
struct shape {
        virtual void draw(); //display the shape
        // ...
};

struct cowboy{
        virtual void draw(); // shoot someone
        //...
};

struct animated_cowboy: shape, cowboy {
        void draw(); // shoot or display?
        //...
};

animated_cowboy player;
shape* shape_ptr=&player;
cowboy* cowboy_ptr=&player;
```

If a function invokes `shape_ptr->draw()`, presumably the intent is to put an image on the screen, not to cause the cowboy to shoot someone!

The generalized overriding proposal allows the derived class to supply separate virtual overrides for the virtual functions inherited from various base classes by substituting a different name for each; the new name will be used in the derived class and all classes derived from it, and the name that would otherwise have been inherited will be undefined in those classes. For instance, the animated_cowboy class could have been defined as follows:

```
struct animated_cowboy: shape, cowboy {
        virtual void render() = shape::draw;
        virtual void draw() = cowboy::draw;
        //...
};
```

A call like `shape_ptr->draw()` will be directed to `animated_cowboy::render()`

Note also that the new name can be the same as the inherited name as in the override specification of `cowboy::draw`. This fact is useful in avoiding potential bugs that otherwise slip past the programmer. Consider the following example:

```
struct B{
        virtual void mf(unsigned int);
        // ...
};

struct D:B{
        void mf(int);
        // ...
};
```

It is likely that `D::mf(int)` was intended to override `B:mf(unsigned int)`. This situation can easily arise when `B` is part of a library and a new version of the library is issued, perhaps changing some of the arguments to virtual functions. While it is possible for a compiler to warn about such a case (current versions of cfront do so), not all compilers will issue a warning, and the construction is legal C++.

The generalized overriding syntax would enable programmers concerned about such potential errors to force the compiler to issue an error diagnostic in case of mismatch. For instance, a careful programmer might code the derived class from the last example as:

```
struct D:B{
        virtual void mf(int) = B::mf;
        // ...
};
```

Since there is no `B::mf(int)` to be overridden, the declaration is in error and a bug is detected at compile time rather than upon execution.

A third use of generalized overriding arises less frequently than the name collision described above and is less difficult to handle via programming, but it is nonetheless reasonable to include the capability in the proposal: sometimes it is desirable to override separate virtual functions in the base classes with a single virtual function in the derived class. The proposal includes the following syntax to allow for this possibility:

```
struct plotter{
        virtual void print(const char*);
        // ...
};

struct screen{
        virtual void display(const char*);
        // ...
};

struct screen_plotter:plotter, screen{
        virtual void output(const char*)={
        plotter::print,
        screen::display
        };
};
```

If the merged derived class virtual function is to be pure, a *0* can be included in the list of base class virtual functions, again underlining the similarity between the syntax for generalized overriding and that of pure virtual functions.

Reprinted from *C++ Journal* Fall 1990

[Editor's Note] The apparent disagreement between Miller above and Koenig in the following on whether an overridden name may be used is just one of the complex issues that surfaced in the email discussions after Miller's article was written.

Andrew Koenig

This proposal has a lot going for it: it solves a real problem; it is a neat extension of syntax that already exists (the syntax for pure virtual functions); it is easy to understand; and it can be implemented efficiently. Indeed, when the proposal was made at the July meeting in Seattle, everyone liked it.

As it happened, the proposal appeared too late for action at that meeting. The final version of this proposal was finished about two days before the meeting, so it would take unanimous consent to vote on it. [The Two-Week Rule was invoked, see Introduction 1.3 Philosophy and Goals] One person did disagree, saying that the folks back home should really have a chance to look at the proposal and comment on it. That blocked any possibility of voting on the proposal in July.

Several people were were annoyed at the time, but the sequel proved that the delay was well justified. A number of questions had been raised while forming the proposal, but it turned out that there were still other questions, some of which were far from easy to answer. One question was whether or not overriding should apply to data members or non-virtual functions. It turns out to be quite easy to cope with those two cases without special language support. For non-virtual functions the technique of writing "forwarding functions" works fine. For data members, one can also use an inline forwarding function in the derived class, if necessary, but this is often unnecessary because it is usually unwise to have public data members anyway. For these reasons, it was proposed that overriding should be allowed only for virtual functions.

The next question was more subtle: does overriding a function remove the overridden name? For example, should the `animated_cowboy` struct in the last example be considered to have a `draw` member at all? What if `draw` were overridden from only a single base class? For example—should this be legal?

```
struct animated_cowboy: public shape, public cowboy {
        //
        public:
                void display() = shape::draw;
                void draw() { //...  };
};
```

The debate went on for quite a while on this question with no conclusive answer. The final consensus was that it was slightly simpler to define if this was made illegal: that is, once `animated_cowboy::draw` is ambiguous, it should not be possible to use overriding to remove that ambiguity. Instead, it should be necessary to choose new names for the `draw` functions inherited from both the base classes.

Shortly before the November meeting. Doug McIlroy suggested a different solution to the original problem:

```
struct shape {
        virtual void draw(); //display the shape
        // ...
};

struct cowboy{
        virtual void draw(); // shoot someone
        //...
};

struct shape_derived: shape {
        virtual void render() { shape::draw(); }
        virtual void draw() { render(); }
        //...
};
```

```
struct cowboy_derived: cowboy {
        virtual void shoot() { cowboy::draw(); }
        virtual void draw() { shoot();}
        //...
};

struct animated_cowboy: shape_derived, cowboy_derived {
        // override render() or shoot() if desired
        // do NOT override draw()
        //...
};

animated_cowboy* cb = new animated_cowboy;
shape* sp = cb;
cowboy* cp = cb;
sp->draw(); // calls animated_cowboy::render();
cp->draw(); // calls animated_cowboy::shoot();
```

For each of our base classes, we have a new class whose sole purpose is to rename the virtual members for which we wish to avoid name clashes. It does this by

- defining a function with a new name that just calls the old function from the base class and
- re-defining the function with the old name to forward calls to the new function.

Once people understood Doug McIlroy's solution, support for the new feature evaporated. It is true that this technique does result in an extra level of virtual function calls, but it should be possible for a compiler to recognize this technique and optimize those calls away. It is probably less work to do that than to implement overriding anyway. Moreover, the advantage of Doug McIlroy's technique is that people can begin using it immediately without waiting for the feature to materialize in their implementations.

The proposal was tabled at the November meeting.

[Epilogue] The Two-Week Rule and the extensive discussions on the email reflectors saved the committee from the embarrassment of approving an unnecessary language feature.