Doc No: SC22/WG21/N1583 = 04-0023

Project: JTC1.22.32

Date: Thursday, February 12, 2004

Author: Francis Glassborow

email: francis@robinton.demon.co.uk

Inheriting Constructors

1 The Problem

Consider:

```
class base {
public:
     base();
     base(base const &);
     base(int);
     ~base();
     base & operator=(base const &);
     void member();
private:
// whatever
};

class derived: public base{};
```

The `derived` either inherits or the compiler implicitly generates functions to match all the `base` class functions with the exception of the non-default constructors. The four special members will work fine; as long as the programmer makes no attempt to define any of them, the compiler will do 'the right thing'. Other member functions are also fine, and if they get hidden the programmer can write a `using`-declaration to bring them into the `derived` scope.

The best that a programmer can do for constructors is to write constructors that forward to the `base` class ones. However as soon as s/he does that s/he also has to define the default versions if those are wanted.

In addition we have a maintenance problem because the owner of `derived` must track the `base` constructors if s/he wishes to continue to have `derived` behave as `base`.

2 The Proposal

I propose that we extend the `using` syntax to:

```
using derived = base;
```

to authorize the compiler to generate constructors for derived that match the constructors of `base` parameters (with base replaced with `derived`). If `derived` has extra data members they will be default initialized as would be the case for the default constructor.

The access level of the `derived` constructors is to be the same as that of the equivalent `base` constructor regardless of whether the inheritance is `public`, `protected` or `private`. It shall not be an error for such a declaration to implicitly declare/define a `private` constructor 'calling' a `private` base constructor though any attempt to use such a constructor, even in the `derived` scope, is ill-formed, diagnostic required. I.e. as is general practice, the error is at the point of use.

A generated constructor based on an explicit qualified base constructor will itself be explicit qualified. If the programmer wishes to overrule that they must explicitly declare and define that constructor.

Should two generated constructors have the same signature the result is ambiguity at the point of call. Where this case could arise the implementor of the derived class should provide an explicit declaration/definition that will be selected in preference to the ambiguous pair.

The programmer may explicitly declare any constructor that would otherwise be compiler generated by the above and optionally provide a definition (of course if no definition is provided then link errors will occur if an attempt is made to use that constructor).

3 Discussion

This proposal uses something akin to the namespace alias type syntax because it effectively is 'renaming' a group of special functions that strictly according to the Standard do not have names.

This proposal is different to class scope using declarations because it involves compiler generated code rather than simple name injection. Perhaps this syntax might make the distinction clearer:

```
default derived = base;
```

By requiring a specific import declaration (a pity import isn't a keyword) code that relies on suppressing constructors by not declaring them in a derived class will continue to work as before.

This proposal just makes constructors largely consistent with other member functions and, in particular, the four special member functions that can be compiler generated today.

Lois Goldthwaite expressed an interest as to how often the problem this proposal seeks to address actually occurs in practice.

Note that there is still an open question about whether access should be capped by an access specifier. My preference is to say 'no'. In other words, use of this mechanism always injects all the base class constructors into the derived class with exactly the same access that they had in the base class. The caveat being that any `private` constructors from the base class are both `private` and uncallable in the derived class.

Templated constructors also need consideration. I am uncertain if there are any subtle implications to inheriting these. I tend to think not but template technology is not one of my strong points.

Question from Anthony Williams

Consider the following:

```
class Derived;
class Base {
public:
    Base(Derived const&);
};

class Derived: public Base
{
public:
    using Derived=Base;
};
```

Is the imported constructor a copy constructor?

This is covered by the paragraph above concerning generated constructors with the same signature. It is ill-formed because both the generated copy constructor and the generated constructor from the non-copy constructor of the base have the same signature.

Possibly the best way to view this proposal is that it aims to provide a mechanism to make all constructors function in the same way. Currently the two default constructors can be compiler generated in the derived class to call the base class constructor (or constructors in the case of multiple inheritance). What this proposal seeks is to allow the programmer to authorize similar behavior for all other base class constructors. While the primary objective is to provide for the simple single inheritance without virtual bases determining the semantics of the more complicated cases can be largely achieved by considering the semantics of the two compiler generated default constructors.

Changes to the Working Paper

These are not provided at this point. It seems more important to agree the mechanism.

However, note that the above proposal only touches on constructors and effectively specifies rules for compiler generation of constructors. It should, therefore, have zero or very low impact on other parts of the WP. We will require to add clauses concerning compiler generation of constructors for derived classes from their base classes.