**Clarification on Programming Task**

**4\_Class\_Design/Visibility1**

This example should explain the rationale behind function hiding as demonstrated by the *Visibility1* programming task. Assume we have the following class hierarchy:

**class Base {**

**public:**

**virtual void foo( void\* );**

**};**

Class Base has a member function foo that takes a parameter of type void\*, and all calls to foo(NULL) are resolved to B::foo(void\*). Let’s assume for a second that there’s no name hiding and that B::foo(void\*) is visible in all its derivates. Let’s also assume that a descendent from Base defines a function foo(int):

**class Derived : public Base {**

**public:**

**void foo( int );**

**};**

Without name hiding, both functions are visible and participating in overload resolution. Which functions will be called in the following example?

**Derived\* d = new Derived;**

**Base\* b = d;**

**b->update( NULL );**

**d->update( NULL );**

The first function call is resolved to B::foo(void\*), the second call however to D::foo(int), since int is a better match for integral zero (i.e. NULL) than any pointer type.

This behavior was deemed undesirable when the language was designed. As a better approach, it was decided to follow the "name hiding" specification, meaning that each class starts with a "clean sheet" with respect to each method name it declares. In order to override this behavior, an explicit action is required from the user: originally a redeclaration of inherited method(s) (currently deprecated), now an explicit use of using-declaration:

**class Derived : public Base {**

**public:**

**using Base::foo;**

**void foo( int );**

**};**

It seems to be reasonable to observe that this behavior violates the IS-A relationship between the classes. However, apparently back then it was decided that in the end name hiding would prove to be a lesser evil.