Doc Number X3J16/91-0144
WG21/N0077

Fixing an Encapsulation Loophole in C++

Alan Snyder

11 July 1990 (DRAFT)

9-Jul/10-Jul/11-Jul

I'm not convinced that there is a problem. We might elaborate the protection system to allow every protection option but that would be a bad idea

Abstract

C++ contains a sophisticated facility for controlling access to class members. This note describes a "loophole" in C++ access control, and suggests a language extension to correct the loophole. The loophole involves the ability of a derived class to override an inherited virtual member function with a definition whose execution is crucial to the integrity of objects of the derived class. The proposed extension supports controlled access to class members via qualified names.

Introduction

A common practice in object-oriented programming is for a child class to extend the definition of a parent class method. The parent class method is extended by redefining the method in the child class; typically, the child class method definition invokes the parent class method and performs some additional computation. We consider the case where the child class is a subtype of the parent class.

For example, one can define a `counting-stack` class that inherits from a `stack` class. `Counting-stack` is a subtype of `stack`. A `counting-stack` is like a `stack`, except it has an additional operation to return the current number of elements. `Counting-stack` redefines the `push` operation to invoke the `push` operation on `stack` and increment a locally-defined counter. (The `pop` operation is redefined in an analogous way.) It is essential for the correct operation of `counting-stacks` that a client be prevented from invoking the `stack` definition of `push` on a `counting-stack` object: to do so would violate the integrity constraint that the value of the local counter be equal to the number of elements on the `stack`.

In current C++, if one defines `counting-stack` as described above, a client cannot be prevented from invoking the `stack` definition of `push` on a `counting-stack` object. Specifically, a derived class that overrides a public virtual member function defined in a public base class cannot prevent a client from invoking the base class member function on a derived class object (using a qualified name, e.g. `stack::push`).

The problem arises only when the base class is a public base class and the virtual member function is public in the base class. Note that mak-

ing the virtual member function protected or private in the derived class (not possible in C++) would not help: the client could still access the base class member function via a qualified name in a base class context, where the member is public.

If the virtual member function is private or protected in the base class, then it is inaccessible to clients of the base class or the derived class, so no problem arises. If the virtual member function is public in the base class, but the base class is a private base class of the derived class, then C++ will prevent the client from viewing a derived class object from a base class context, so no problem arises.

The example illustrates the value of preventing client access even in the case of public members of public base classes.

The problem arises only for virtual member functions, not for data members or non-virtual member functions. Subject to access control, a non-virtual base class member can always be accessed on a derived class object simply by viewing the object from the base class context. Only virtual member functions can be overridden such that the derived class definition is seen even in the base class context.

Example

The following example illustrates the problem:

```
class B {
    public:
      virtual void f (int);
    };
class D : public B {
    public:
      void f (int); // D::f overrides B::f
    };

void B::f (int) {;}
void D::f (int i) {B::f (i+1);}

void test1 (B& b, D& d) {

  b.B::f (1); // call 1
  d.B::f (2); // call 2
  d.D::f (3); // call 3
  b.f    (4); // call 4
  d.f    (5); // call 5

  }
```

This program is legal in current C++. We would like to define `B` and `D` such that call 2 in the above example is illegal. To be effective, we must

also make call 1 illegal, as the variable `b` could denote the same object as variable `d`.

The proposed solution is to allow the base class to declare different accesses for the member function definition (`B::f`) and the virtual member function (`f`). We want the virtual member function (`f`) to be public, but the member function definition (`B::f`) to be protected. We draw a distinction, not currently emphasized in C++, between the qualified name (`B::f`) and the unqualified name (`f`). The qualified name refers to the actual function definition, the unqualified name refers to the virtual member that indirectly denotes the function definition (or an overriding function definition).

The Proposal

Our specific proposal is as follows. We propose that C++ be extended to permit an access declaration to name a (previously declared) member of the class containing the access declarations (as opposed to a base class member, as currently permitted). Such an access declaration defines the permitted access to the member using a qualified name. Access to the member using an unqualified name is unchanged by the presence of the access declaration. It does not make sense for a client to have greater access via a qualified name than via an unqualified name, so it is illegal to use the access declaration to increase access.

The example using this proposed solution is shown in the following figure. Note that we have chosen to protect `D::f` as well, for the benefit of classes derived from `D`.

No additional security results from using this extension to restrict access to non-virtual members via qualified names: if the member would have been accessible via the qualified name without use of this extension, it could still be accessed even using this extension via the unqualified name from the base class context. Nevertheless, the extension can be used on non-virtual class members to enforce the style guideline mentioned on page 210 of the Annotated C++ Reference Manual: "As a rule of thumb, explicit qualification should be used only to access base class members from a member of a derived class." As in the above example, this effect is achieved by declaring the qualified name to be protected.

Default Access via Qualified Names

The remaining issue is what the default access should be to class members via qualified names. If compatibility with current C++ is paramount, then the default should be that access to a member via the qualified name is the same as access via the unqualified name. However, good software engineering practice suggests that qualified member names be protected. Therefore, it makes sense to make qualified mem-

```
class B {
    public:
      virtual void f (int);
    protected:
      B::f;   // limit access via qualified name
    };
class D : public B {
    public:
      void f (int); // D::f overrides B::f
    protected:
      D::f;   // limit access via qualified name
    };

void B::f (int) {;}
void D::f (int i) {B::f (i+1);}

void test1 (B& b, D& d) {

  b.B::f (1); // call 1 (illegal by definition of B)
  d.B::f (2); // call 2 (illegal by definition of B)
  d.D::f (3); // call 3 (illegal by definition of D)
  b.f    (4); // call 4
  d.f    (5); // call 5

  }
```

ber names protected by default. Although such a default would be incompatible with current C++, I propose that it be given serious consideration. After all, if C++ is successful, then only a small fraction of all C++ programs have yet been written!