A Note on the Definition of Visibility for Protected Class Members in C++

Alberto Nava B., Enzo Chiariotti
Departamento de Computación y
Tecnología de la Información
Universidad Simón Bolívar
Apartado 80659, Caracas 1070-A, Venezuela
emsca!usb!beto@sun.com
emsca!usb!enzo@sun.com

Resumen

En este artículo se muestra como la definición de la visibililidad de los miembros protegidos en el lenguaje de programación C++ causa ciertos problemas en el modelaje de sistemas. Se presentan algunos ejemplos que abstraen situaciones de la "vida real" en los cuales se puede observar como la ocurrencia de dichos problemas afecta la reusabilidad del software. Finalmente se proponen dos definiciones alternas que no sufren de tales inconvenientes y que permitirían mejorar este lenguaje para la programación orientada por objetos.

Abstract

This article shows some problems with the definition of visibility of protected members in the C++ programming language. We present some examples which abstract "real life" situations in which these problems affect code reusability. Finally, we propose two alternative definitions of visibility rules for protected members that not suffer of such problems.

Introduction

Protected fields in C++ [1] are a useful mechanism for partial data hiding while keeping efficiency in implementations. Briefly, if class B is derived from class A then protected members of class A are visible to class B objects. A class designer can define some members as protected, granting access to derived class implementors while at the same time hiding those members from end users.

Typical candidates to be protected are those members which have to deal with object data structure. We want to hide them from end users since we don't want user code to depend on the data structure of the class implementation. However, we also want to provide efficient access to data structures to the implementors of derived

classes. A further use of protected members is to provide some operations to the designers of derived classes that should be hidden from the end user. We could even define a class in which all constructors are protected, so that users cannot declare objects of the class but can use its derived classes.

As Snyder says [3], a class interface can be considered as an agreement with other users: public members are for everyone, protected members are only for implementors of new derived classes and private members should be only for the designer of the original class.

```
class A {
public:
...        // members for all end users
protected:
...        // members for derived class implementors
private:
...        // private stuff you should never know
}
```

Note that a change in the protected interface will affect derived classes but not end users. This is one of the reasons for which Snyder state that a class interface is like an agreement. "Protected" is an agreement between class designer and derived class designers.

Problems with protected fields in C++

Despite the above, the usefulness of protected fields in C++ is to some extent compromised by the following characteristic of the definition:

"A friend or a member function of a derived class can access any public or protected static member of a publicly derived class. A friend or a member function of a derived class can access a protected non-static member of a base class only through a pointer to, reference to, and object of the derived class of which it is friend or member (and pointers to, and objects of classes publicly derived from this class)" [2]. As a result of this, non-static protected members of a parameter of the same class as the base class are not accesible by a friend or a member function of the derived class.

Suppose we want to model bird relationships and we know that *Emperor Penguins* don't like animals either too different from them or from other habitat. Moreover, *Emperor Penguins* don't like *King Penguins* :

```
class Bird {
public:
     virtual int Likes(Bird& other);
     virtual int CanFly() { return 1; }
     void  Show();
protected:
     String Habitat;
     String WhoAmI;
     Photo  Picture;
};


class Emperor_Penguin : public Bird {
public:
     Emperor_Penguin() {
        Habitat = "Antartic"; WhoAmI = "Emperor_Penguin";
        Picture.Load("Emperor_Penguin.map");
     }
     int CanFly()  { return 0; }
     int Likes(Bird& other) {
          return other.Habitat == Habitat &&
                 Picture.Diff(other.Picture) < TOO_DIFFERENT &&
                 !(other.WhoAmI == "King_Penguin")
     }
}
```

Appealing as it may seem, the above code will not compile: function `Likes()` of class `Emperor_Penguin` receives a reference to a `Bird` as a parameter; the body of the function makes use of the protected members `Habitat`, `Picture` and `WhoAmI` which is not allowed by the AT&T definition of C++ [2,73].

We can patch the problem in the above example by making `Emperor_Penguin` a friend class of `Bird`.

```
class Bird {
     friend class Emperor_Penguin;
     friend class King_Penguin;
public:
     virtual int Likes(Bird& other);
     virtual int CanFly() { return 1; }
     virtual void Show();
private:
     String Habitat;
     String WhoAmI;
     Photo  Picture;
};


class Emperor_Penguin : public Bird {
...       // same as above
};
```

However, as Stroustrup says [1,9], use of friends should be avoided whenever possible. Moreover, each time you add a new class derived from `Bird` you have to make that class a friend class of `Bird`.

Another possibility is to define, for each protected member, a public function which returns a reference to it:

```
class Bird {
public:
     virtual int Likes(Bird& other);
     virtual int CanFly() { return 1; }
     virtual void Show();
     String& Habitat() {return _Habitat;}  // Maybe const String&
     String& WhoAmI()  {return _WhoAmI;}
     Photo&  Picture() {return _Picture;}
private:
     String _Habitat;
     String _WhoAmI;
     Photo  _Picture;
};
```

```
class Emperor_Penguin : public Bird {
public:
     Emperor_Penguin() {
        Habitat = "Antartic"; WhoAmI = "Emperor_Penguin";
        Picture.Load("Emperor_Penguin.map");
     }
     int CanFly()  { return 0; }
     int Likes(Bird& other) {
          return other.Habitat() == Habitat &&
                 Picture.Diff(other.Picture()) < TOO_DIFFERENT &&
                 !(other.WhoAmI() == "King_Emperor")
     }
}
```

However, this solution presents some problems:

- When defining these functions in the public interface, their names, arguments and return value types are now part of the agreement between the implementor and the end user. As an example, the return type of `Picture()` , the type `Photo`, is now part of the agreement with the end user. A change in the type used to represent a bird picture (the type of `_Picture`) can affect user code that uses the return value of `Picture()` for some purpose.

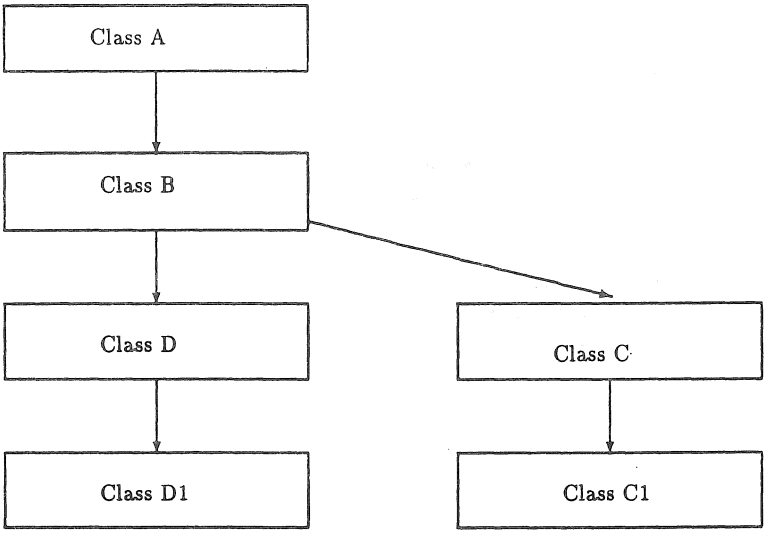
- The introduction of the `WhoAmI()` operation in the public interface goes against the object oriented style of programming, since it encourages users to write code which uses "type-field" solutions [1,9][1,199].

Given the unsatisfactory nature of the solutions proposed, we suggest that a different policy for protected member visibility be defined, to make the first version above correct.

Two Alternative Definitions

A possible alternative definition could be: "A friend or member function of a derived class D can access any public or protected member of a base class B through a pointer to, reference to, or object of class B (and pointers to, references to, or objects of classes publicly derived from B)".

Although this definition is very simple, some objections could be raised regarding its permissiveness. As an example, consider the visibility of protected members from the point of view of friends and member functions of class D in the following hierarchy tree:



According to the AT&T definition a member or friend of class D can access protected non-static members of A and B only through instances of D and D1. On the other hand, our previous definition allows access to protected members of A and B through instances of any class in the tree hierarchy. To date we are not sure whether permit access to protected members of A and B through instances of C and C1 is a good idea or not.

Another possible definition would be to allow access to those members only through instances of classes in the path of direct inheritance; in our example this would imply that class D members could access protected members of A and B through instances of A, B, D and D1, but not of C and C1.

Conclusions

C++ tries to achieve a good balance between many conflicting requirements; not an easy task. However we think that the definition of visibility of protected members in the AT&T definition is not in harmony with other aspects of the language and that some practical examples are affected by this. We suggest that the prohibition on the use of protected members of other instances of the base class must be lifted. This seems to be more consistent with other characteristics of C++.

References

[1] Bjarne Stroustrup. *The C++ Programming Language*, Addison-Wesley, 1986.

[2] Unix System V AT&T C++ Language System, Release 2.0.

[3] Alan Snyder. "Encapsulation and Inheritance in Object-Oriented Programming Languages". OOPSLA '86 Proceedings, September 1986.