N2006=06-0076
2006-04-12
Daveed Vandevoorde
daveed@vandevoorde.com

Accessibility and Visibility in C++ Modules

1 Introduction

In C++ today ("C++03" and earlier variants), the notions of accessibility and visibility are independent. Members of classes and namespaces are visible whenever they are "in scope" and there is no mechanism to reduce this visibility from the point of declaration. Accessibility is only a parameter for class members and is orthogonal to the notion of visibility. This latter observation is frequently surprising to novice C++ programmers in examples like the following[1]:

```
#include <stdlib.h>
struct Base {
  // ...
private:
  void exit();
};
struct Derived: Base {
  void quit() {
    exit(0); // Error: Finds private (but visible)
  }          // Base::exit.
};
```

More significantly, the inability to altogether hide private members is a hindrance to encapsulation: Changing the private details of a class can inadvertently (and sometimes subtly) affect the correctness of client code.
The proposal to introduce modules in C++ (N1964) also proposes to make private members invisible outside the module in which they are declared. The example above may then be recast as follows:

```
// File 1:
export namespace Lib {
public:
  struct Base {
    // ...
  private:
    void exit();  // Invisible outside Lib
  };
}
```

[1] The example is based on a real-life case that caused substantial grief.

```
// File 2:
#include <stdlib.h>
import namespace Lib;
struct Derived: Lib::Base {
  void quit() {
    exit(0);  // Okay: Finds ::exit.
  }
};
```

The module proposal also allows namespace scope members of modules to be declared private (or public).
While the visibility rule outlined above provides the desired results in the majority of situations, there are some significant cases in which invisibility is not desirable for private members. For example:

```
namespace N {
  struct S {
    void f(int);
  private:
    void f(long);
  };
}
void g(N::S &s) {
  s.f(50000000L);  // Access error
}
```

Here a private member is used to catch a potentially problematic narrowing conversion. Making the member invisible would defeat its purpose. N1964 therefore proposes a new access specifier **`prohibited`** to indicate that a member cannot be used.

```
export namespace P {
public:
  void f(double) { ... }
prohibited:
  void f(int);
}
```

Discussion in the evolution group in Berlin (April 2006) indicated there is an interest in seeing alternatives to this approach to the problem. This paper is a survey of such possible alternatives. An opinion as to the desirability of the various options is also offered.

2 Alternatives

2.1 Classic private + new internal access

A fairly straightforward alternative to the approach in N1964 is to keep the meaning of **private** to be "visible but not accessible" in all scopes, and to introduce a new access specifier to indicate members that are invisible outside their module. The introductory example could then be rewritten as follows:

```
// File 1:
export namespace Lib {
public:
  struct Base {
    // ...
  intern:
    void exit();  // Invisible outside Lib
  };
}

// File 2:
#include <stdlib.h>
import namespace Lib;
struct Derived: Lib::Base {
  void quit() {
    exit(0);  // Okay: Finds ::exit.
  }
};
```

The access specifier **intern** could instead be spelled **hidden**, **invisible**, **internal**, **cloaked**, **concealed**, **masked**, **secret**, or **unseen**. The identifiers **hidden** and **invisible** are unfortunately fairly frequently used in existing C and C++ code.

The potential advantage of this alternative is that the classic meaning of private is retained in the module world, which may ease a transition from a header based library implementation to a module-based implementation. With this alternative the need for the specifier **prohibited** is less obvious.

2.2 Classic private + orthogonal visibility

Accessibility and visibility need not be tied into a single keyword. Instead, the two could be specified independently:

```
// File 1:
export namespace Lib {
public:
  struct Base {
    // ...
  intern private:
    void exit();  // Invisible outside Lib
  };
}

// File 2:
#include <stdlib.h>
import namespace Lib;
struct Derived: Lib::Base {
  void quit() {
    exit(0);  // Okay: Finds ::exit.
  }
}
```

The potential advantage of this approach is that a module would have the ability to define public or protected members that can only be used within the module (they would be **`intern public`** or **`intern protected`**).

2.3 Heuristic visibility

A third alternative that has been considered is the idea to use a heuristic to decide on the visibility of a member. Specifically, is it the case that it would be sufficient to make nonfunction members invisible outside the module?
Unfortunately, that appears not to be the case. In particular, the majority of the reported cases of interference from private members involve member functions (as is the case in the introductory example).

3 Opinion

I slightly favor the approach of N1964 over the alternative "Classic private + new cloaked access", because

- I expect most existing private members will want to be invisible in a module-based environment (hence less code will need to change to achieve the best possible interface).
- The new access specifier **`prohibited`** is more specific than **`private`**. For example, prohibited members cannot be used even in private code and providing a body or initializer for a prohibited member cannot be meaningful (and can therefore be diagnosed).
- I find the aesthetic aspects of the vocabulary public/protected/private/prohibited are more pleasing than the alternative.

The alternative of making accessibility and visibility entirely orthogonal is less appealing to me because it adds more complexity than it solves real problems. The advantages it occasionally offers can largely be emulated in the other models using friend declarations and/or helper classes.
The approach of making private members invisible based on their kind is syntactically the most pleasing, but I don't think there is a rule that would address existing issues to a sufficient extent.

4 Acknowledgments

Thanks to Lois Goldthwaite, Alisdair Meredith, Sam Saariste, Bronek Kozicki, Lawrence Crowl, and Benjamin Kosnik for useful input and ideas regarding the issue studied in this paper.