CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 17
March 31, 2016

Name Visibility

Demo: Craps Game

Name Visibility

Private derivation (default)

`class B : A { ... };` specifies private derivation of `B` from `A`.

A class member inherited from `A` become `private` in `B`.
Like other private members, it is inaccessible outside of `B`.

If `public` in `A`, it can be accessed from within `A` or `B` or via an instance of `A`, but not via an instance of `B`.

If `private` in `A`, it can only be accessed from within `A`.
It cannot even be accessed from within `B`.

Private derivation example

Example:

```
class A {
private:  int x;
public:   int y;
};
class B : A {
    ... f() {... x++; ...} // privacy violation
};
//-------- outside of class definitions --------
A a; B b;
a.x   // privacy violation
a.y   // ok
b.x   // privacy violation
b.y   // privacy violation
```

Public derivation

`class B : public A { ... };` specifies public derivation of `B` from `A`.

A class member inherited from `A` retains its privacy status from `A`.

If `public` in `A`, it can be accessed from within `B` and also via instances of `A` or `B`.

If `private` in `A`, it can only be accessed from within `A`.
It cannot even be accessed from within `B`.

Public derivation example

Example:

```
class A {
private:  int x;
public:   int y;
};
class B : public A {
    ... f() {... x++; ...} // privacy violation
};
//-------- outside of class definitions --------
A a; B b;
a.x   // privacy violation
a.y   // ok
b.x   // privacy violation
b.y   // ok
```

The protected keyword

`protected` is a privacy status between `public` and `private`.

Protected class members are inaccessible from outside the class (like `private`) but accessible within a derived class (like `public`).

Example:

```
class A {
protected: int z;
};
class B : A {
    ... f() {... z++; ...} // ok
};
```

Protected derivation

`class B : protected A { ... };` specifies protected derivation of `B` from `A`.

A `public` or `protected` class member inherited from `A` becomes `protected` in `B`.

If `public` in `A`, it can be accessed from within `B` and also via instances of `A` but not via instances of `B`.

If `protected` in `A`, it can be accessed from within `A` or `B` but not from outside.

If `private` in `A`, it can only be accessed from within `A`.
It cannot be accessed from within `B`.

Surprising example 1

```
1   class A {
2   protected:
3     int x;
4   };
5   class B : public A {
6   public:
7     int f() { return x; }          //  ok
8     int g(A* a) { return a->x; }  //  privacy violation
9   };
```

Result:

```
tryme1.cpp: In member function 'int B::g(A*)':
tryme1.cpp:3: error: 'int A::x' is protected
tryme1.cpp:9: error: within this context
```

Surprising example 2: contrast the following

```
1   class A { };
2   class B : public A {};    // <-- public derivation
3   int main() { A* ap; B* bp;
4     ap = bp; }
```

Result: OK.

```
1   class A { };
2   class B : private A {};    // <-- private derivation
3   int main() { A* ap; B* bp;
4     ap = bp; }
```

Result:

```
tryme2.cpp: In function 'int main()':
tryme2.cpp:4: error: 'A' is an inaccessible base of 'B'
```

Surprising example 3

```
1   class A { protected: int x; };
2   class B : protected A {};
3   int main() { A* ap; B* bp;
4     ap = bp; }
```

Result:

```
tryme3.cpp: In function 'int main()':
tryme3.cpp:4: error: 'A' is an inaccessible base of 'B'
```

Names, Members, and Contexts

Data and function names can be declared in many different **contexts** in C++: in a class, globally, in function parameter lists, and in code blocks (viz. local variables).

Often the same identifier will be declared multiple times in different contexts.

Two steps to determining the meaning of an occurrence of an identifier:

1. Determine which declaration it refers to.
2. Determine its accessibility according to the privacy rules.

Declaration and reference contexts

Every reference `x` to a class data or function member has two contexts associated with it:

- The **declaration context** is the context in which the referent of `x` (the thing that `x` refers to) appears.
- The **reference context** is the context in which the reference `x` appears.

Accessibility rules apply to class data and function members depend on both the declaration context and the reference context of a reference `x`.

Declaration context example

Example:

```
int x = 3;                    // declaration context: global
class A {
  int x;                      // declaration context: A
  void f(int x) {...}         // declaration context: parameter
  void g() {int x; ... }      // declaration context: block local
};
```

Reference context example

```
class A {
  int x;
  int f() {return x;}          // reference context A
  int g(A* p) {return p->x;}   // reference context A
};
int main() {
  A obj;
  obj.x;                       // reference context global
}
```

All three commented occurrences of `x` have declaration context `A` because all three refer to `A::x`, the data member declared in class `A`.

Inside and outside class references

A reference `x` to a data/function member of class `A` is

- inside class `A` if the reference context of `x` is `A`;
- outside class `A` otherwise.

For simple classes:

- an inside reference `x` is always valid.
- an outside reference `x` is valid iff the referent is public.

Examples

References to A::x

```
class A {
   int x;
   int f() { return x; }          // inside
   int g(A* p) { return p->x; }   // inside
   int h();
};

int A::h () { return x; }         // inside

#include <iostream>
int main() {
  A aObject;
  std::cout << aObject.x;         // outside
};
```

Inherited names

In a derived class, names from the base class are inherited by the derived class, but their privacy settings are altered as described in the last lecture.

The result is that **the same member exists in both classes** but with possibly different privacy settings.

Question: Which privacy setting is used to determine visibility?

Answer: The one of the declaration context of the referent.

Inheritance example

```
class A { protected: int x; };
class B : private A {
  int f() { return x; }          // ok, x is inside B
  int g(A* p) { return p->x; }  // not okay, x is outside A
};
```

Let `bb` be an instance of class `B`. Then `bb` contains a field `x`, inherited from class `A`. This field has two names `A::x` and `B::x`.

The names are distinct and may have different privacy attributes. In this example, `A::x` is protected and `B::x` is private.

First reference is okay since the declaration context of `x` is `B`. Second reference is not since the declaration context of `x` is `A`. Both occurrences have reference context `B`.

Inaccessible base class

A base class pointer can only reference an object of a derived class if doing so would not violate the derived class's privacy. Recall surprising example 2 (bottom):

```
1   class A { };
2   class B : private A {};    // <-- private derivation
3   int main() { A* ap; B* bp;
4     ap = bp; }
```

The idea is that with private derivation, the fact that `B` is derived from `A` should be completely invisible from the outside.

With protected derivation, it should be completely invisible except to its descendants.

Visibility rules

Every class member has one of four **privacy attributes**: *public*, *protected*, *private*, or *hidden*.

These attributes determine the locations from which a class member can be seen.

- *public* members can be seen from any location.
- *protected* members can be seen from inside the class or its children.
- *private* members can only be seen from inside the class.
- *hidden* members cannot be seen at all.

Explicit privacy attributes

The privacy attributes for declared class members are given explicitly by the privacy keywords `public`, `protected`, and `private`.

There is no way to explicitly declare a hidden member.

Example:

```
class A {
private:   int x;
protected: int y;
public:    int z;
};
```

Implicit privacy attributes

Inherited class members are assigned implicit privacy attributes based on their attributes in the parent class and by the kind of derivation, whether `public`, `protected`, or `private`.

1. If the member is *public* in the parent class, then its attribute in the child class is given by the kind of derivation.
2. If the member is *protected* in the parent class, then its attribute in the child class is *protected* for public and protected derivation, and *private* for private derivation.
3. If the member is *private* or *hidden* in the parent class, then it is *hidden* in the child class.

Implicit privacy chart

Below is a revision of the chart presented in lecture 10.

Attribute in base class	Kind of Derivation: public	Kind of Derivation: protected	Kind of Derivation: private
public	public	protected	private
protected	protected	protected	private
private	hidden	hidden	hidden
hidden	hidden	hidden	hidden

Attribute in derived class.

Summary

1. All members of the base class are inherited by the derived class and appear in every instantiation of that class.
2. All inherited members receive implicitly defined privacy attributes.
3. Visibility of all data members is determined solely by their privacy attributes.
4. Public and protected base class variables are always visible within a derived class.
5. Private and hidden base class variables are never visible in the derived class.
6. The kind of derivation never affects the visibility of inherited members in the derived class; only their implicit attributes.

Demo: Craps Game

Game Rules

The player (known as the *shooter*) rolls a pair of fair dice

1. If the sum is 7 or 11 on the first throw, the shooter wins; this event is called a natural.
2. If the sum is 2, 3, or 12 on the first throw, the shooter loses; this event is called craps.
3. If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, this number becomes the shooter's point. The shooter continues rolling the dice until either she rolls the point again (in which case she wins) or rolls a 7 (in which case she loses).

(From http://www.math.uah.edu/stat/games/Craps.html)

A Craps simulator

Demo 17-Craps illustrates the use of derived classes in order to allow the simulator to work with both random dice and "prerecorded" dice throws stored in a file.