CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 19
November 9, 2016

Name Visibility (continued)

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

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

Result: OK.

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

Result:

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

Surprising example 3

```
class A { protected: int x; };
class B : protected A {};
int main() { A* ap; B* bp;
  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 above.

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. Their declaration contexts are A and B, respectively.

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

Inheritance example (continued)

```
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
};
```

To determine whether a reference to `x` is legal, one must first decide which `x` is being referenced.

First `x` reference refers to `B::x`. Second `x` reference refers to `A::x`. Both occurrences have reference context `B`.

First reference is okay since declaration and reference contexts are the same. Second reference is not okay since `A::x` is protected and the reference context, `B`, is outside of `A`.

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):

```
class A { };
class B : private A {};     // <-- private derivation
int main() { A* ap; B* bp;
  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.

		Kind of Derivation		
		public	protected	private
Attribute in base class	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.

Linear Data Structure Demo

Using polymorphism

Similar data structures:

- Linked list implementation of a stack of items.
- Linked list implementation of a queue of items.

Both support a common interface:

- `void      put(Item*)`
- `Item*     pop()`
- `Item*     peek()`
- `ostream& print(ostream&)`

They differ only in where `put()` places a new item.

The demo `19-Virtual` (from Chapter 15 of textbook) shows how to exploit this commonality.

Interface file

We define this common interface by the abstract class.

```
class Container {
  public:
    virtual void      put(Item*)       =0;
    virtual Item*     pop()            =0;
    virtual Item*     peek()           =0;
    virtual ostream& print(ostream&) =0;
};
```

Any class derived from it is required to implement these four functions.

We could derive `Stack` and `Queue` directly from `Container`, but we instead exploit even more commonality between these two classes.

Class Linear

```
class Linear: public Container {
  protected:  Cell* head;
  private:    Cell* here; Cell* prior;
  protected:  Linear();
    virtual   ~Linear ();
              void     reset();
              bool     end() const;
              void     operator ++();
    virtual   void     insert( Cell* cp );
    virtual   void     focus() = 0;
              Cell*    remove();
              void     setPrior(Cell* cp);
  public:     void     put(Item * ep);
              Item*    pop();
              Item*    peek();
    virtual   ostream& print( ostream& out );
};
```

Example: Stack

```
class Stack : public Linear {
  public:
    Stack(){}
    ~Stack(){}
    void  insert( Cell* cp ) { reset(); Linear::insert(cp); }
    void  focus(){ reset(); }

    ostream& print( ostream& out ){
        out << "  The stack contains:\n";
        return Linear::print( out );
    }
};
```

Example: Queue

```
class Queue : public Linear {
  private:
    Cell*   tail;

  public:
    Queue() { tail = head; }
    ~Queue(){}

    void  insert( Cell* cp ) {
        setPrior(tail); Linear::insert(cp); tail=cp; }
    void  focus(){ reset(); }
};
```

Class structure

Class structure.

- `Container` specifies the common interface.
- `Linear` contains the bulk of the code. It is derived from `Container`.
- `Stack` and `Queue` are both derived from `Linear`.
- `Cell` is a "helper" class that is aggregated by `Linear`.
- `Item` is the base type for the container elements. It is defined by a `typedef` here but would normally be specified by a template.
- `Exam` is a non-trivial item type used by `main` to illustrate stacks and queues.

C++ features

The demo illustrates several C++ features.

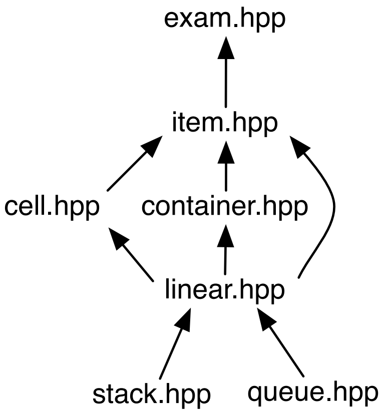
1. [`Container`] Pure abstract class.
2. [`Cell`] Friend functions.
3. [`Cell`] Printing a pointer in hex.
4. [`Cell`] Operator extension `operator Item*()`.
5. [`Linear`] Virtual functions and polymorphism.
6. [`Linear`] Scanner pairs (prior, here) for traversing a linked list.
7. [`Linear`] Operator extension `operator ++()`
8. [`Linear`, `Exam`] Use of `private, protected, and public` in same class.

#include structure

Getting #include's in the right order.

Problem: Making sure compiler sees symbol definitions before they are used.

Partial solution: Make dependency graph. If not cyclic, each .hpp file includes the .hpp files just above it.


exam.hpp
item.hpp
cell.hpp
container.hpp
linear.hpp
stack.hpp
queue.hpp