CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 11
October 6, 2011

Name Visibility Revisited

Interacting Classes and UML

Name Visibility Revisited

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` 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.

Interacting Classes and UML

What is a Class: Syntax

Mine	other relevant information
- name1: type1 - name2: type2 + name5: type5	data members
+ Mine (param list)	constructors
+ ~Mine()	destructor
- funA(param list): type3 - funB(param list): type4 + funC(param list): type6 + funD(param list): type7	function members

Can include global operators in such a diagram, adding a 4th row.

Class Relationships

- Class relationship studies the connectivity of a set of classes in an OO system.
- A large-scale OO system can have a large number of classes
  - .Net 8000 classes (.Net CF: 1400 classes)
- There are empirical studies on the relationship between class relationship metric and software quality, e.g., A Validation of Object-Oriented Design Metrics as Quality Indicators, 1996.

Class Relationship Between Two Classes

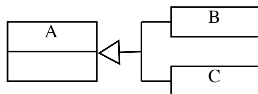
Scenarios in which class `B` appears in definition of class `A`?

Class B appears in Definition of Class A

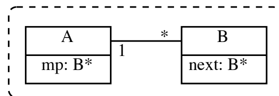
Class B is related with class A

- A derives from B
- B is a friend of A
- definition of class B is nested in class A

UML diagram for derivation:



UML diagram for friend:



B as Data Members in A

Class B objects as data members in class A, e.g.,

- `B b;`
- `B b[2];`
- `B* bp01; // one object`
- `B* bpm; // multiple B objects`

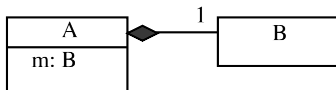
These reflect different class relationships:

- **composition**: the first two cases. It is also referred to as **has-a-value** relation.
- **association**: the next two cases. It is also referred to as **has-a-reference** relation.

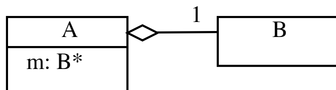
The association relationship is weaker. A special type of association is called **aggregation**: when a B object is a "part" of an A object.

B as Data Members in A

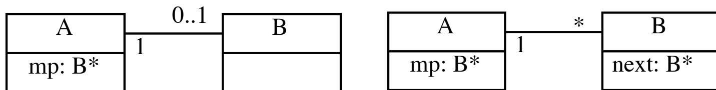
UML diagram for **composition**:



UML diagram for **aggregation**:



UML diagrams for **simple association** (left) and **one-many association** (right):



Creation and Deletion

Identifying **composition**, **association** and **aggregation** helps with object creation and deletion:

- **composition**: creation/deletion of `B` objects is part of the creation/deletion of `A`. Compiler will automatically create (invoking default constructor) `B` objects when creating `A` object.
- **aggregation**: generally, if a `B` object is aggregated into a single `A` object, then the `A` object is responsible to create and delete the `B` object.

Example: `BarGraph` Class Interaction

What is the class diagram of the `BarGraph` program?

What types of relationships do we identify? Why?