CIT237

Chapter 15: Inheritance (Part 2)

November 6, 2019

Reminders

- Monday, November 11 is a holiday:
  the College is closed that day – no classes.
- On Tuesday, November 12, the MONDAY class schedule will be followed.

- Quiz 6 will be held at the start of class on
  Wednesday, November 20.
- The material covered on Quiz 6 will be:
  - Lectures of October 28 through November 13.
  - Chapters 15, 16 and 17.
- Project 3:
  - We will discuss Project 3 in class today.
  - The due date is December 2.

Recall Previous Lecture

- Inheritance: provides a way to create a new class from an existing class.
- The new class is a specialized version of the existing class:
  - A class called “Vehicle” could represent attributes and behaviors which are common to all vehicles.
  - A class called “Car” could be considered a “specialization” of “Vehicle”: it has all attributes and behaviors of “Vehicle”, plus other attributes and behaviors which may be unique to “Car”.

Inheritance Terminology

- Base class: the class which is inherited from.
  - Sometimes called the “Parent” class.
  - Sometimes called the “Superclass”
- Derived class: inherits from the base class
  - Sometimes called the “Child” class.
  - Sometimes called the “Subclass”

Inheritance Syntax

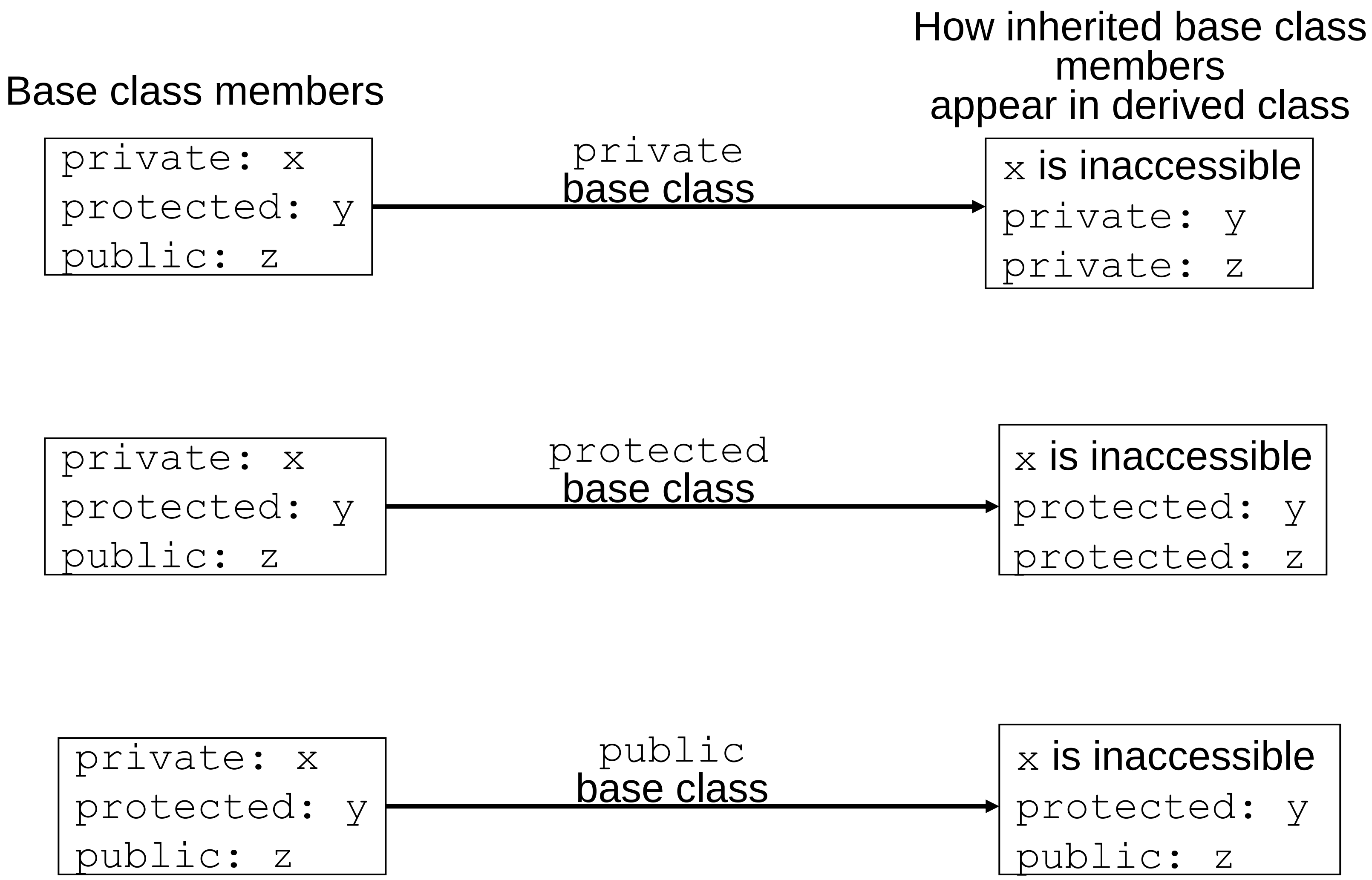
```
class Student         // base class
{
   . . .
};
class UnderGrad : public student
{                   // derived class
   . . .
};
```

Class Access Specifiers

Determine how members of the derived class can access members of the base class:

1) `public` – object of derived class can be treated as object of base class (not vice-versa)
2) `protected` – more restrictive than `public`, but allows derived classes to know details of parents
3) `private` – prevents objects of derived class from being treated as objects of base class. (This is the default.)

Inheritance vs. Access



Base class members

How inherited base class members appear in derived class

private: x
protected: y
public: z

private base class

x is inaccessible
private: y
private: z

private: x
protected: y
public: z

protected base class

x is inaccessible
protected: y
protected: z

private: x
protected: y
public: z

public base class

x is inaccessible
protected: y
public: z

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class’s constructor is executed first, followed by the derived class’s constructor
- When an object of a derived class is destroyed, its destructor runs first, followed by the destructor for the base class.

Redefining Base Class Functions

- Redefining function: function in a derived class that has the *same name and parameter list* as a function in the base class
- Typically used to replace a function in base class with different actions in derived class

Redefining is NOT Overloading

- Not the same as overloading – with overloading, parameter lists must be different

- Objects of base class use base class version of function; objects of derived class use derived class version of function

Base Class Example: **GradedActivity**

```
class GradedActivity   {
protected:
   char letter;
   double score;
   void determineGrade();
public:
   GradedActivity(){ letter=' '; score=0.0;}
   void setScore(double s) {
       score = s;
       determineGrade(); }
   double getScore() const {
       return score;
    }
   char getLetterGrade() const {
       return letter; }
};
```

Note `setScore` function

Derived Class Example: `CurvedActivity`

```
class CurvedActivity : public GradedActivity
{
protected:
   double rawScore;      // Unadjusted score
   double percentage;    // Curve percentage
public:
   CurvedActivity() : GradedActivity() {
       rawScore=0.0; percentage=0.0;
   }
   void setScore(double s) {
      rawScore = s;
      GradedActivity::
            setScore(rawScore*percentage);
   }
   . . .
};
```

Redefined `setScore` function

Program 15-7: CurvedActivity

- **CurvedActivity** is a subclass of **GradedActivity**.
- The **GradedActivity** version of **setScore()** enforces the “traditional” version of setting a letter grade.
- The **CurvedActivity** version of **setScore()** adjusts the raw score by a user-selected percentage, resulting in a different letter grade.

Problem with Redefining

- When a subclass redefines a function, there can be a situation where the base class version of the function is executed when the programmer really wanted the derived class version.
- The sample program:

  Ch15_sample_code_StaticBindingExample

  illustrates this problem:
- An object of the subclass (CurvedActivity) does not get to run its own version of the setScore member function.

Static Binding

- If a Subclass has a function with the same signature (name and parameters) as one in the SuperClass, then the system needs a way to decide which one to use.
- When the default “static” binding is used, which function gets called depends on the type of the object *variable* used in the function call.
- But the same object, invoked with a different object variable, may result in different behavior.

Dynamic Binding

- What we would *usually* prefer is for our program to call the function associated with the actual *object* instance (not the <u>type</u> of the *variable*).
- Make the member function a *virtual* member function:
  - Add the keyword **`virtual`** to the declaration of the function in the Base Class (the parent class).
- Replace

```
void setScore(…);
```

with

```
virtual void setScore(…);
```

Polymorphism and Virtual Member Functions

- Virtual member function: function in base class that expects to be redefined in derived class
- Function defined with key word `virtual`:

  `virtual void` *functionName* `() {...}`
- Supports dynamic binding: functions bound at run time to function that they call
- Without virtual member functions, C++ uses static (compile time) binding

Virtual Functions

- A virtual function is dynamically bound to calls at runtime.

- At *runtime*, C++ determines the type of object making the call, and binds the function to the appropriate version of the function.

Virtual Functions -- Syntax

- To make a function virtual, place the virtual key word before the return type in the base class's declaration:

```
virtual char getLetterGrade() const;
```

- The compiler will not bind the function to calls. Instead, the program will bind them at runtime.

BindingDemo.cpp

- This program attempts to demonstrate the problem with “static binding” and why “dynamic binding” can resolve the problem. Consider this situation:
- Two classes: **BaseClass** and **DerivedClass**.
  - The "DerivedClass" class is a subclass of "BaseClass".
  - Each of these classes has a function named "**nonVirtualFunction**", and a function named "**virtualFunction**", but ONLY **BaseClass** has a function named "**x**".
  - DerivedClass::nonVirtualFunction **“redefines”** BaseClass::nonVirtualFunction.
  - DerivedClass::virtualFunction **“overrides”** BaseClass::virtualFunction.

Polymorphism Requires References or Pointers

- Polymorphic behavior is only possible when an object is referenced by a reference variable or a pointer.

Redefining vs. Overriding

- In C++, redefined functions are statically bound and overridden functions are dynamically bound.

- So, a virtual function is overridden, and a non-virtual function is redefined.

Virtual Destructors

- It's a good idea to make destructors virtual if the class could ever become a base class.
- Otherwise, the compiler will perform static binding on the destructor if the class ever is derived from.
- See Program 15-14 for an example

Abstract Base Classes and Pure Virtual Functions

- Pure virtual function: a virtual member function that must be overridden in a derived class that has objects
- Abstract base class contains at least one pure virtual function:

```
virtual void Y() = 0;
```

- The `= 0` indicates a pure virtual function
- Must have no function definition in the base class

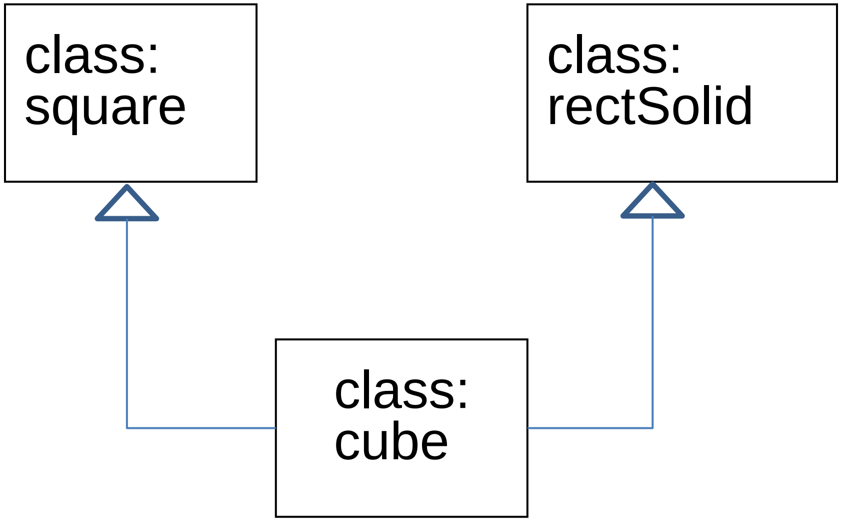
Abstract Base Classes and Pure Virtual Functions

- Abstract base class: class that can have no objects. Serves as a basis for derived classes that may/will have objects
- A class becomes an abstract base class when one or more of its member functions is a pure virtual function

Multiple Inheritance

- A derived class can have more than one base class
- Each base class can have its own access specification in derived class's definition:

```
   class cube : public square,
                public rectSolid;
```



Multiple Inheritance and Constructor Invocation

- Arguments can be passed to both base classes' constructors:

```
cube::cube(int side) :
square(side),
        rectSolid(side, side, side);
```

- Base class constructors are called in order given in class declaration, not in order used in class constructor

Multiple Inheritance -- Problems

- Problem: what if base classes have member variables/functions with the same name?
- Solutions:
  - Derived class redefines the multiply-defined function
  - Derived class invokes member function in a particular base class using scope resolution operator ::
- Compiler errors occur if derived class uses base class function without one of these solutions

Inheritance: Summary

- Constructor invocation
- Static Binding: compile-time function selection.
- Dynamic Binding: run-time function selection.
- Redefining vs. Overriding Base Class Functions
  - Do not confuse with “overloading”.