COP 3331
OBJECT ORIENTED DESIGN
SPRING 2017

WEEK 9: INHERITANCE AND POLYMORPHISM

SCHINNEL SMALL

USF
UNIVERSITY OF SOUTH FLORIDA

INHERITANCE

INHERITANCE

- Inheritance provides a way to create a new class from an existing class (called the *base class,* or *parent*)
  - It's purpose is to reduce complexity
- The new class (called the *derived class* or *child*) is a specialized version of the base class
- The derived class automatically inherits all of the member variables and functions
  - Traditionally, the new class did not inherit the constructor or destructor; C++ 11 made it optional to inherit some constructors from the base class

INHERITANCE

- Inheritance establishes the “is a” relationship between objects
  - A poodle is a dog
  - A car is a vehicle
  - A flower is a plant
  - A football player is an athlete

- When an “is a” relationship exists between classes, the specialized class has all of the characteristics of a general class, plus additional characteristics that makes it special

INHERITANCE

- Inheritance may be:
  - Single: derived class has a single base class
    - A base class can also be derived from another class
  - Multiple: derived class has more than one base class

- Objects of the derived class has:
  - All members defined in base class
  - All members defined in derived class

- Objects of the derived class can use:
  - All public members defined in derived class
  - All public members declared in base class

INHERITANCE

- Syntax:

```
class Student          // base class
{
  . . .
};

class UnderGrad : public Student
{// derived class (with base class
 // reference)
  . . .
};
```

INHERITANCE EXAMPLE

- See:
  - GradedActivity.h
  - GradedActivity.cpp
  - FinalExam.h
  - FinalExam.cpp
  - main.cpp

INHERITANCE

- `class FinalExam : public GradedActivity`
  - FinalExam is derived from GradedActivity
  - public indicates the *base class access specification*
    - This determines how the base class members appear in the derived class
    - In this case the public members of GradedActivity will become public members of FinalExam
  - Private members of GradedActivity is inherited but "invisible" to the code of FinalExam
    - Can only be accessed by GradedActivity

INHERITANCE

- The constructors of GradedActivity are not listed in the FinalExam class
  - Recall: the constructors set up objects of the class in which they are defined

- Since the function getScore and getLetterGrade are inherited, they can be accessed like any public member

INHERITANCE

- Note: Inheritance does not work in reverse!
  - It is not possible for a base class to call a member function of a derived class

- This will not work:

```
class BadBase
{
    private:
        int x;
    public:
        BadBase() { x = getVal(); }   // Error!
};
```

```
class Derived : public BadBase
{
    private:
        int y;
    public:
        Derived(int z) { y = z; }
        int getVal() { return y; }
};
```

PROTECTED CLASS MEMBERS
BASIC CLASS ACCESS SPECIFICATION

PROTECTED CLASS MEMBERS

- Protected members of a base class are like private members, but they may be accessed by functions in derived classes

- Protected members remain inaccessible to the rest of the program

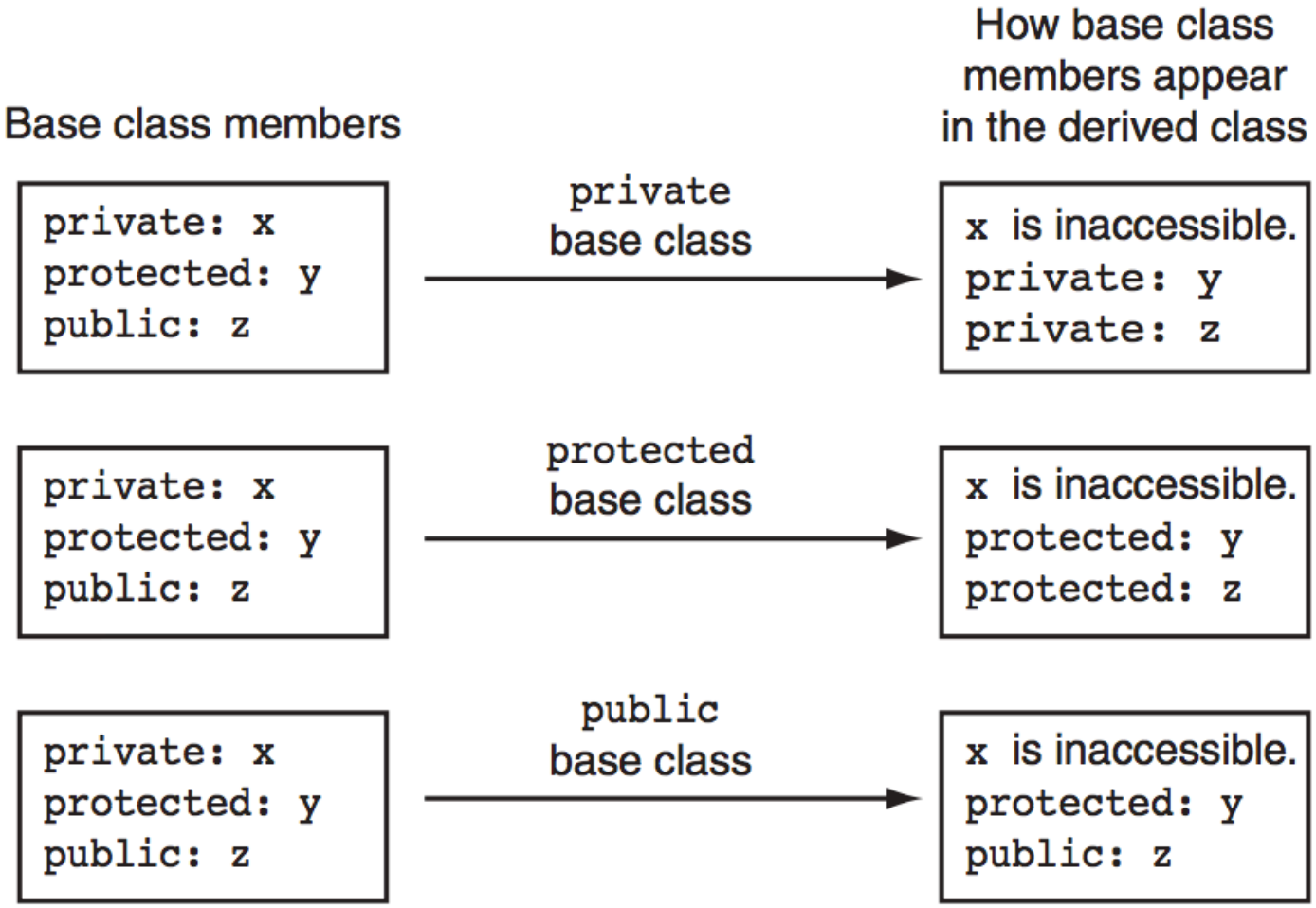
- To make a member protected, change the `private` keyword to `protected`

BASIC CLASS ACCESS SPECIFICATION

- Do not confuse basic class access specification with member access specification
  - Member class specification determines how members *defined within the class* are accessed
  - Base class specification determines how *inherited members* are accessed

- If a base class specification is left out the default access specification is private
  - `class Test: Grade`

BASE ACCESS CLASS SPECIFICATION



INHERITANCE: CONSTRUCTORS AND DESTRUCTORS

CONSTRUCTORS AND DESTRUCTORS

- Derived classes can have their own constructors and destructors
- When an object is created, the base class's constructor is called before the derived class's constructor
- When an object is destroyed, the destructors are called in reverse order;
  - derived class constructors are are called first, then base class destructors

EXAMPLE

```
#include <iostream>
using namespace std;
class BaseClass
{
public:
    BaseClass()  // Constructor
    { cout << "This is the BaseClass constructor.\n"; }
    ~BaseClass() // Destructor
    { cout << "This is the BaseClass destructor.\n"; }
};

class DerivedClass : public BaseClass
{
public:
    DerivedClass()  // Constructor
    { cout << "This is the DerivedClass constructor.\n"; }
    ~DerivedClass()  // Destructor
    { cout << "This is the DerivedClass destructor.\n"; }
};

int main()
{
    cout << "We will now define a DerivedClass object.\n";
    DerivedClass object;
    cout << "The program is now going to end.\n”;
}
```

Program Output

```
We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.
```

CONSTRUCTORS AND DESTRUCTORS

- If a base class's constructor takes arguments (or if there is more than one constructor in the base class), the derived class constructor can pass arguments to it

- Notation
  - `ClassName (Param List) : BaseClassName(List)`
  - Outside class definition:

```
ClassName::ClassName (Param List) : BaseClassName(List)
```

INHERITANCE CONSTRUCTOR EXAMPLE

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
private:
   double width;
   double length;
public:
   // Default constructor
   Rectangle()
      { width = 0.0;
        length = 0.0; }

   // Constructor #2
   Rectangle(double w, double len)
      { width = w;
        length = len; }

   double getWidth() const
      { return width; }

   double getLength() const
      { return length; }

   double getArea() const
      { return width * length; }
};
#endif
```

```
#ifndef CUBE_H
#define CUBE_H
#include "Rectangle.h"

class Cube : public Rectangle
{
protected:
   double height;
   double volume;
public:
   // Default constructor
   Cube() : Rectangle()
      { height = 0.0; volume = 0.0; }

   // Constructor #2
   Cube(double w, double len, double h) : Rectangle(w, len)
      { height = h;
        volume = getArea() * h; }

   double getHeight() const
      { return height; }

   double getVolume() const
      { return volume; }
};
#endif
```

CONSTRUCTOR INHERITANCE

- C++11 provides a way for derived class to inherit *some* of the base class's constructors
  - A derived class cannot inherit the default constructor or copy constructor

- This is useful when a derived class simply invokes the base class constructor

CONSTRUCTOR INHERITANCE

- Consider: in this example, the myDerived constructor simple calls the (respective) myBase constructor

```
class MyBase
{
private:
    int ival;
    double dval;
public:
    MyBase (int i)
    { ival = i; }
    MyBase (double d)
    { dval = d; }
};
```

```
class MyDerived:MyBase
{
public:
    MyDerived (int i): MyBase (i)
    {}
    MyDerived double d): MyBase (d)
    {}
};
```

CONSTRUCTOR INHERITANCE

- C++ 11 allows us to rewrite the MyDerived class as:

```
class MyDerived:MyBase
{
    using MyBase::MyBase;
};
```

- Therefore, the following statement causes the MyDerived class to call the MyBase class's constructors
  - `MyDerived d1(22);`
  - `MyDerived d2(3.79);`

CONSTRUCTOR INHERITANCE

- A derived class can still have its own constructors and inherit from the base class

- However, if a derived class constructor has the same parameter list as a base class constructor, the base class constructor will not be inherited

REDEFINING BASE CLASS FUNCTIONS

REDEFINING BASE CLASS FUNCTIONS

- Sometimes, it is helpful to overload a base class function with a function of the same name in the derived class

- This is not the same as function overloading!
  - Overloaded functions have the same name as other functions, but the parameter lists are different
  - Redefined functions occur when a derived class function has the same name has the base class function
    - Parameter lists can be the same because the derived class function is always called by objects of the derived class type

REDEFINED FUNCTION EXAMPLE

This is a modified version of Graded Activity

```
class GradedActivity
{
protected:
   char letter;              // To hold the letter grade
   double score;             // To hold the numeric score
   void determineGrade();    // Determines the letter grade
public:
   // Default constructor
   GradedActivity()
      { letter = ' '; score = 0.0; }

   // Mutator function
   void setScore(double s)
      { score = s;
        determineGrade();}

   // Accessor functions
   double getScore() const
      { return score; }

   char getLetterGrade() const
      { return letter; }
};
```

Note `setScore` function

REDEFINED FUNCTION EXAMPLE

```
1  #ifndef CURVEDACTIVITY_H
2  #define CURVEDACTIVITY_H
3  #include "GradedActivity.h"
4
5  class CurvedActivity : public GradedActivity
6  {
7  protected:
8     double rawScore;      // Unadjusted score
9     double percentage;    // Curve percentage
10 public:
11    // Default constructor
12    CurvedActivity() : GradedActivity()
13       { rawScore = 0.0; percentage = 0.0; }
14
15    // Mutator functions
16    void setScore(double s)
17       { rawScore = s;
18         GradedActivity::setScore(rawScore * percentage); }
19
20    void setPercentage(double c)
21       { percentage = c; }
22
23    // Accessor functions
24    double getPercentage() const
25       { return percentage; }
26
27    double getRawScore() const
28       { return rawScore; }
29 };
30 #endif
```

Redefined `setScore` function

REDEFINING BASE CLASS FUNCTIONS

- The line

```
GradedActivity::setScore(rawScore*percentage);
```

  calls the base class's version of setScore with the expression passed as an argument

- This is often used to extend a class or give it additional capabilities

PROBLEM WITH REDEFINING

- Consider this situation:
  - Class `BaseClass` defines functions `x()` and `y()`, where `x()` calls `y()`
  - Class `DerivedClass` inherits from `BaseClass` and redefines function `y()`
  - An object `D` of class `DerivedClass` is created and function `x()` is called
  - When `x()` is called, which `y()` is used, the one defined in `BaseClass` or the the redefined one in `DerivedClass`?

PROBLEM WITH REDEFINING

BaseClass

```
void X();
void Y();
```

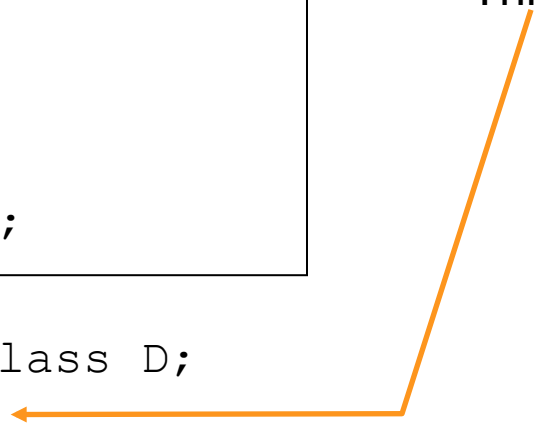
DerivedClass

```
void Y();
```

```
DerivedClass D;
D.X();
```

Object `D` invokes function `X()` in `BaseClass`
Function `X()` invokes function `Y()` in `BaseClass` and not function `Y()` in `DerivedClass`

Function calls are bound at compile time
This is static binding



POLYMORPHISM AND VIRTUAL FUNCTIONS

POLYMORPHISM

- Polymorphism is the ability for code or operations to behave differently in different contexts
  - We've already seen them in C++ with function and operator overloading

- When used in C++ specifically, polymorphism allows an object variable (or pointer) to reference objects of different types, and *to call the correct member functions, based on the object being referenced*

POLYMORPHISM

- Recall: in our last example we saw that the function y() in the base class was being called, even though:
  - We created an object of the derived class
  - the derived class has its own version of y()

- This was determined at compile time due to our inheritance of the base class and it's use of x()

- We can use virtual functions to remedy this situation

VIRTUAL FUNCTION

- A virtual function is a member function that is *dynamically bound* to function calls

- In dynamic binding, C++ determines which function to call at runtime, based on the type of object responsible for the call

- Virtual functions are declared by using the virtual keyword before the return type in the base class's function declaration

VIRTUAL FUNCTION

```
class GradedActivity
{
protected:
   double score;    // To hold the numeric score
public:
   // Default constructor
   GradedActivity()
      { score = 0.0; }

   // Constructor
   GradedActivity(double s)
      { score = s; }

   // Mutator function
   void setScore(double s)
      { score = s; }

   // Accessor functions
   double getScore() const
      { return score; }

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

Back to the original version of Graded Activity

The function is now virtual.

The function also becomes virtual in all derived classes automatically!

POLYMORPHISM AND VIRTUAL FUNCTIONS

- Although all derived classes of a virtual function become virtual automatically, it is a good idea to use the virtual keyword in the derived classes (for documentation purposes)

- Note: polymorphism *requires references to an object or pointers*
  - Not possible to pass by value

- You can define a pointer to a base class and assign it the address of a derived class object

POLYMORPHISM AND VIRTUAL FUNCTIONS

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2); }
};
```

```
int main ()
{
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
}
```

BASE CLASS POINTERS

- Base class pointers and references only know about members of the base class
  - You can't use a bass class pointer to call a derived class fuction

- Redefined functions in derived classes will be ignored unless the base class declares the function virtual

REDEFINING VS. OVERRIDING

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

- Therefore, virtual functions are overriden, while non-virtual functions are redefined

- C++ 11 allows you to explicitly state when virtual functions should and should not be overridden with the `override` and `final` keywords

REDEFINING VS. OVERRIDING

```
#include <iostream>
using namespace std;

class Base
{
public:
      virtual void functionA(int arg) const
      { cout << "This is Base::functionA" << endl; }
};

class Derived : public Base
{
public:
      virtual void functionA(long arg) const
      { cout << "This is Derived::functionA" << endl; }
};

int main()
{
      // Allocate instances of the Derived class.
      Base *b = new Derived();
      Derived *d = new Derived();

      // Call functionA with the two pointers.
      b->functionA(99);
      d->functionA(99);
}
```

Program Output

```
This is Base::functionA
This is Derived::functionA
```

REDEFINING VS. OVERRIDING

```
#include <iostream>
using namespace std;

class Base
{
public:
      virtual void functionA(int arg) const
      { cout << "This is Base::functionA" << endl; }
};

class Derived : public Base
{
public:
      virtual void functionA(int arg) const override
      { cout << "This is Derived::functionA" << endl; }
};

int main()
{
      // Allocate instances of the Derived class.
      Base *b = new Derived();
      Derived *d = new Derived();

      // Call functionA with the two pointers.
      b->functionA(99);
      d->functionA(99);
}
```

Program Output

```
This is Derived::functionA

This is Derived::functionA
```

REDEFINING VS. OVERRIDING

- Sometimes you want to prevent a virtual member function from being overridden and further

- C++ 11 allows you to use final in the following manner:

```
virtual void message() const final;
```

- Any attempts to override this function will generate an error

ANNOUNCEMENTS

ANNOUNCEMENTS

- HW 4 (to be) posted on Canvas

- Have a great spring break! ☺