Advanced Object Oriented Programming

Inheritance and aggregation

Seokhee Jeon
Department of Computer Engineering
Kyung Hee University
jeon@khu.ac.kr
**Inheritance?**

- Extend a class to create a new class while retaining the members of the original class and adding new members.

- Inheritance is an *is-a* relationship.

- A *derived class* is-*a* base class with more detail added.
Example - Polygons

Triangle

Perimeter: \( a + b + c \)
Area: \( \sqrt{\frac{\text{perimeter}}{2}} \times \left( \frac{\text{perimeter}}{2} - a \right) \times \left( \frac{\text{perimeter}}{2} - b \right) \times \left( \frac{\text{perimeter}}{2} - c \right) \)

Square

Perimeter: \( 4 \times a \)
Area: \( a^2 \)

Rectangle

Perimeter: \( 2 \times (a + b) \)
Area: \( a \times b \)
Example: Common attributes

- Common attributes for polygons
  - Perimeter
  - Area
Example: Common attributes

- Common attributes for polygons
  - Perimeter
  - Area

- Make a class using common attributes!!
  - Base Class

Base Class

Polygon
Example: Base Class - Polygon

class Polygons
{
    protected:
        double area;
        double perimeter;

        void printArea () const;
        void printPeri () const;
}; // Class Polygons
Derived classes

- Unique attributes for each polygon (triangle, square, rectangle)
  - How to calculate Perimeter
  - How to calculate Area

- Make derived classes with unique parameters from base class
Example: Base and derived classes

- Polygon (Base Class)
  - Triangle (Derived Class)
  - Square (Derived Class)
  - Rectangle (Derived Class)
Example: Base and derived classes

Class *derivedClassName* : *inheritance_type* *baseClassName*
Example: Derived classes for Polygons

class Triangle : public Polygons
{
    private:
    double sideA;
    double sideB;
    double sideC;
    void calcArea ();
    void calcPeri ();
}; // Class Triangle

class Square : public Polygons
{
    private:
    double side;
    void calcArea ();
    void calcPeri ();
}; // Class Square

class Rectangle : public Polygons
{
    private:
    double sideA;
    double sideB;
    void calcArea ();
    void calcPeri ();
}; // Class Rectangle

perimeter = a + b + c
area = sqrt { (perimeter / 2) * [(perimeter / 2) - a] * [(perimeter / 2) - b] * [(perimeter / 2) - c] }

perimeter = 4 * a
area = a²

perimeter = 2 * (a + b)
area = a * b
Example: Derived class object

```cpp
class Triangle : public Polygons {
    private:
        double sideA;
        double sideB;
        double sideC;
        void calcArea();
        void calcPeri();
}; // Class Triangle
```

```cpp
Triangle triangleObject;
```

```cpp
class Square : public Polygons {
    private:
        double side;
        void calcArea();
        void calcPeri();
}; // Class Square
```

```cpp
Square squareObject;
```

```cpp
class Rectangle : public Polygons {
    private:
        double sideA;
        double sideB;
        void calcArea();
        void calcPeri();
}; // Class Rectangle
```

```cpp
Rectangle rectangleObject;
```
Example: Derived class object

What is inherited?

- Every data
- Every func. except:
  - Constructor
  - Destructor
  - Nonmember func.
  - Assignment operator
  - Virtual method
Constructors, destructors, nonmember functions, assignment operators, and virtual methods are not inherited in a derived class. If needed, they must be created.
# Inheritance access rules

<table>
<thead>
<tr>
<th>Base Class Type</th>
<th>Inheritance Type</th>
<th>Derived Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td><strong>Private</strong></td>
<td>Inherited but inaccessible Public</td>
</tr>
<tr>
<td>Protected</td>
<td></td>
<td>Private</td>
</tr>
<tr>
<td>Public</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private</td>
<td><strong>Protected</strong></td>
<td>Inherited but inaccessible Protected</td>
</tr>
<tr>
<td>Protected</td>
<td></td>
<td>Protected</td>
</tr>
<tr>
<td>Public</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private</td>
<td><strong>Public</strong></td>
<td>Inherited but inaccessible Public</td>
</tr>
<tr>
<td>Protected</td>
<td></td>
<td>Public</td>
</tr>
<tr>
<td>Public</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Inheritance types

Fruit
- count
# price
+ print()

Access:
- private
# protected
+ public

Apple
- price
- print()

private inheritance

Banana
# price
# print()

protected inheritance

Orange
# price
+ print()

public inheritance
class Polygons {
    protected:
        double area;
        double perimeter;
    public:
        Polygons () {};
        ~Polygons () {};
        void printArea () const;
        void printPeri () const;
}; // Class Polygons

/* ====== Polygons :: printArea ======
Prints the area of a polygon.
Pre    area calculated & stored in area
Post   area printed    */
void Polygons :: printArea () const
{
    cout << "The area of your polygon is "
         << area << endl;
    return;
} // Polygons printArea

/* ====== Polygons :: printPeri ======
Prints the perimeter of a polygon.
Pre    polygon perimeter calculated and stored
Post   perimeter printed    */
void Polygons :: printPeri () const
{
    cout << "The perimeter of your polygon is "
         << perimeter << endl;
    return;
} // Polygons printPeri
Triangle class code

```cpp
class Triangle : public Polygons {
    private:
        double sideA;
        double sideB;
        double sideC;

        void calcArea();
        void calcPeri();

    public:
        // initialization constructor
        Triangle (double sideAIn,
                  double sideBIn,
                  double sideCIn);
}; // Class Triangle

Triangle :: Triangle (double sideAIn,
                       double sideBIn, double sideCIn)
{
    // Verify sides are valid
    if  ( ((sideAIn + sideBIn) <= sideCIn)
        || ((sideBIn + sideCIn) <= sideAIn)
        || ((sideCIn + sideAIn) <= sideBIn) )
    {
        cout << "Invalid Triangle\n";
        exit (100);
    } // if
    // Valid Triangle
    sideA = sideAIn;
    sideB = sideBIn;
    sideC = sideCIn;

    calcPeri();
    calcArea();
    return ;
} // Triangle initialization constructor

void Triangle :: calcArea ()
{
    double halfPeri = perimeter / 2;
    area = (  halfPeri
              * (halfPeri - sideA)
              * (halfPeri - sideB)
              * (halfPeri - sideC) );
    area = sqrt(area);
    return;
} // Triangle calcArea

void Triangle :: calcPeri ()
{
    perimeter = sideA + sideB + sideC;
    return;
} // Triangle calcPeri
```
Main function code for Triangle Class

```cpp
#include <iostream>
#include <cmath>
#include <cstdlib>
using namespace std;

#include "p12-03.h"     // Polygon class
#include "p12-04.h"     // Triangle class

int main ()
{
    cout << "Start Polygon Demonstration\n\n";
    Triangle tri (3, 4, 5);
    tri.printPeri();
    tri.printArea();
    cout << "\nEnd Polygon Demonstration\n";
    return 0 ;
} // main
```
Another example: EmpInfo inheritance

### Derived Classes

<table>
<thead>
<tr>
<th>Derived Class</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>class EmpInfo : <strong>private</strong> SecurityEmp</td>
<td></td>
</tr>
</tbody>
</table>
| | ```
| private:
| int ID;
| ``` |
| | ```
| protected:
| int Salary;
| ``` |
| | ```
| public:
| char Name[200];
| ``` |
| | |
| class EmpInfo : **protected** ImportantEmp |
| | ```
| private:
| int ID;
| ``` |
| | ```
| protected:
| int Salary;
| ``` |
| | ```
| public:
| char Name[200];
| ``` |
| | |
| class EmpInfo : **public** RegularEmp |
| | ```
| private:
| int ID;
| ``` |
| | ```
| protected:
| int Salary;
| ``` |
| | ```
| public:
| char Name[200];
| ``` |
TIP! Overriding access specifier

```cpp
class B
{
    protected:
        int datum;
    public:
        void print();
        ...
}; // B
```

```cpp
class D : private B
{
    protected:
        using B::datum;
        ...
}; // D
```

‘datum’ is protected member variable in class D
Constructor of derived class

- Constructor calling?
  - Constructor of base class invoked implicitly **before** the constructor of derived class
  - You can give parameters to the constructor of base class

```cpp
Derived :: Derived (parameters) : BaseClass (parameters)
{
    ...
}
```
Destructor of derived class

- Destructor calling?
  - Destructor of base class invoked implicitly after the destructor of derived class
Example: Employee Class
Example: Employee Class

class Employee {
    protected:
        int id;

    public:
        Employee (int idIn);
    ~Employee ();
};  // Employee

Employee :: Employee (int idIn)
{
    id = idIn;
    cout << "Base constr for Employee: "
        << id << endl;
}
// Employee constructor

Employee :: ~Employee ()
{
    cout << "Base distr for Employee: "
        << id << endl;
}
// Employee Destructor

class SalaryEmp : public Employee {
    private:
        int salary;

    public:
        SalaryEmp (int idIn, int salaryIn);
    ~SalaryEmp ();
};  // SalaryEmp

SalaryEmp :: SalaryEmp (int idIn, int salaryIn)
    : Employee (idIn)
{
    salary = salaryIn;
    cout << "Deriv constr for SalaryEmp: "
        << id << " Salary: " << salary << endl;
}
// SalaryEmp constructor

SalaryEmp :: ~SalaryEmp ()
{
    cout << "Deriv distr for SalaryEmp: "
        << id << endl;
}
// SalaryEmp destructor

class HourlyEmp : public Employee {
    private:
        float payRate;
        float hours;

    public:
        HourlyEmp (int idIn,
                    float payRateIn, float hoursIn);
    ~HourlyEmp ();
};  // HourlyEmp Class

HourlyEmp :: HourlyEmp (int idIn,
                        float payRateIn, float hoursIn)
    : Employee (idIn)
{
    payRate = payRateIn;  //Derived data
    hours = hoursIn;
    cout << "Deriv constr for HourlyEmp: "
        << id << " Pay Rate: " << payRate
        << " Hours: " << hours << endl;
}
// HourlyEmp constructor

HourlyEmp :: ~HourlyEmp ()
{
    cout << "Deriv distr for HourlyEmp: "
        << id << endl;
}
// HourlyEmp destructor
Example: Employee Class

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

#include "p12-09.h"

int main ()
{
    SalaryEmp slryEmp (1234, 43000);
    HourlyEmp hrlyEmp (5678, 15.76, 40);

    return 0;
} // main

/* Results
Base constr for Employee : 1234
Derv constr for SalaryEmp: 1234 Salary: 43000
Base constr for Employee : 5678
Derv constr for HourlyEmp: 5678 Pay Rate: 15.76 Hours: 40
Derv destr for HourlyEmp: 5678
Base destr for Employee : 5678
Derv destr for SalaryEmp: 1234
Base destr for Employee : 1234
*/
```
Copy constructor of derived class

• Pass the parameter, which is a reference to the existing derived object, to the constructor of the base class

```cpp
Derived::Derived (const Derived& derivedObject)
    : Base(derivedObject)
{
    ...
    // Derived Part Initialization
}
```

The reference type of a base class can also be used to refer to an object of a derived class
Overriding

- Overriding?
  - Redefining the member function (of base class) in the derived class

- Difference between Overriding and Overloading?
  - Overloading
    - Same function name,
    - but different parameter list
  - Overriding
    - Same function name,
    - and same parameter list
Overriding Example

- Overriding “print()” at derived class

```cpp
Void Derived :: print()
{
    Base::print(); // print base class data

    cout << “Datum in Derived Class is: ” << derivedDatum << endl;
}
```

- If required, you can call original “print()” of base class

```cpp
Derived dObject;
dObject.Base::print();
```
Overriding Example

```cpp
int main ()
{
    BaseClass baseObj (1);
    Derived   derObj (12, 13);

    cout << "Using print in base class: \n";
    baseObj.print ();

    cout << "\nUsing print in derived class: \n";
    derObj.print ();

    return 0;
} // main
```

/* Results
Using print in base class:
**Datum in Base Class is: 1

Using print in derived class:
**Datum in Base Class is: 12
**Datum in Derived Class is: 13
*/
Polymorphism: overloading

- Allow many different forms with the same name; The meaning is determined by the context

- Example in real life
  - "open" a door
  - "open" a jar
  - "open" a book

- Operator/function overloading is an simple example of polymorphism
Polymorphism with inheritance

- Primary use of polymorphism in object-oriented programming
- write several versions of a function, each in a separate class related by inheritance
- When called, the function appropriate for the object being referenced is executed
- The programmer (and the program) does not have to know the exact type of the object in advance, and so the exact behavior is determined at run-time (this is called late binding or dynamic binding). (from Wikipedia)
Pointing Derived Class using Pointer: Static Binding

- **Static Binding?**
  - Function is bound to the pointer at compile time.
  - Although, the object (to which it points) is dynamically allocated by the `new` operator.

```cpp
BaseClass* objPtr = new BaseClass();
```

```cpp
objPtr = new DervClass();
```

Because `objPtr` is statically bound, `BaseClass::print` is called.
#include <iostream>
using namespace std;

class BaseClass
{
    public:
        void print() const {cout << "Base class object\n";}
}; // BaseClass

class DervClass: public BaseClass
{
    public:
        void print() const {cout << "Derived class object\n";}
}; // DervClass

int main()
{
    BaseClass* objPtr = new BaseClass();
    objPtr->print();
    delete objPtr;       // Release memory

    objPtr = new DervClass();  // Reallocate object
    objPtr->print();
    return 0;
} // main

/*  Results
 Base class object
 Base class object */
**Pointing Derived Class using Pointer: Dynamic Binding**

- **Dynamic Binding?**
  - ‘Late binding’ until run time: `function type definition key word “Virtual”`
  - Compiler delay binding until the program is run

```
BaseClass* objPtr = new BaseClass();

objPtr = new DervClass();
```

Because `objPtr` is dynamically bound, `DervClass::print` is called.
```
#include <iostream>
using namespace std;

class BaseClass
{
    public:
        virtual void print() const
            {cout << "Base class object\n";}
}; // BaseClass

class DerivClass: public BaseClass
{
    public:
        virtual void print() const
            {cout << "Derived class object\n";}
}; // DerivClass

int main ()
{
    BaseClass* objPtr = new BaseClass ();
    objPtr->print ();
    delete objPtr; // Release memory

    objPtr = new DerivClass ();
    objPtr->print ();
    return 0;
} // main

/* Results
Base class object
Derived class object */
```
Problem for Destructor & Virtual Destructor

BaseClass
{
    private:
        int* p;
    public:
        BaseClass() {p = new int[4];}
        ~BaseClass() {delete[] p;}
    }  // BaseClass

DervClass
{
    private:
        float* q;
    public:
        DervClass() {q = new float[4];}
        ~DervClass() {delete[] q;}
    }  // DervClass
...

int main ()
{
    BaseClass* objPtr = new DervClass();
    ...
    delete objPtr;
    ...

(a) After instantiation

(b) After delete
Virtual Destructor

- **Type**
  
  ```
  virtual ~BaseClass() { ... }
  ```

- **Sequence**
  
  - Destructor of derived class
  - Destructor of base class
Model is required

- The virtual functions do not force the overriding of a function
- There are occasions that we want to force the derived class to override (redefine) a function
  - What if in the future we want more polygons?
  - We need to define a model in the base class and let the derived classes follow the model
- In the base class, we can define the minimum number of functions and the format that is needed for each derived class to include
Pure Virtual Function

- Syntax
  
  ```cpp
  virtual return_type identifier (parameter list) = 0
  ```

- Declared in a base class
- Have no action in base class (just for conceptual requirement)
- Should be defined in each derived class
Abstract Class

- **Class with at least one pure virtual member function**
  - Can not make a object using Abstract Class
  - Should be used as a base class for derived classes

- **Why use Abstract Class?**
  - When, required to define common class for multiple classes
    (e.g. polygon as a base class for derived triangle, rectangle, square)
  - However, it has no meaning, and just a conceptual class
class Polygons
{
   protected:
   double area;
   double perimeter;
   virtual void calcArea () = 0;
   virtual void calcPeri () = 0;
   public:
   Polygons () {}
   ~Polygons (){}
   void printArea () const;
   void printPeri () const;
};  // Class Polygons

void Polygons :: printArea () const
{
   cout.setf(ios::fixed, ios :: floatfield);
   cout.precision(1);
   cout << "Area of polygon: " << setw (5)
        << area << endl;
   return;
}  // Polygons printArea

void Polygons :: printPeri () const
{
   cout.setf(ios::fixed, ios :: floatfield);
   cout.precision(1);
   cout << "Perimeter of polygon: " << setw (5)
        << perimeter << endl;
   return;
}  // Polygons printPeri

• calcArea() and calcPeri() moved from derived classes to base class.
• we can invoke these func. using Polygon*,
   whether it is triangle, square or rectangle.
   → Polymorphism
   → new additional derived classes does not impact to conventional codes
Example: Modified Triangle Class

class Triangle : public Polygons
{
    private:
        double sideA;
        double sideB;
        double sideC;

        virtual void calcArea();
        virtual void calcPeri();

    public:
        Triangle (double sideAIn, double sideBIn,
                 double sideCIn);
};  // Class Triangle

Triangle :: Triangle (double sideAIn, double sideBIn,
                     double sideCIn)
{
    // Verify sides are valid
    if ( ((sideAIn + sideBIn) <= sideCIn)
        || ((sideBIn + sideCIn) <= sideAIn)
        || ((sideCIn + sideAIn) <= sideBIn))
    {
        cout << "Invalid Triangle\n";
        exit (100);
    } // if

    // Valid Triangle
    sideA = sideAIn;
    sideB = sideBIn;
    sideC = sideCIn;
    calcPeri();
    calcArea();
    return;
} // Triangle constructor

void Triangle :: calcArea()
{
    double halfPeri = perimeter / 2;
    area = ( halfPeri
              * (halfPeri - sideA)
              * (halfPeri - sideB)
              * (halfPeri - sideC) );
    area = sqrt(area);
    return;
} // Triangle calcArea

void Triangle :: calcPeri()
{
    perimeter = sideA + sideB + sideC;
    return;
} // Triangle calcPeri
Example: Modified Rectangle Class

class Rectangle : public Polygons
{
    private:
        double sideA;
        double sideB;
    virtual void calcArea();
    virtual void calcPeri();

    public:
        Rectangle(double sideAIn, double sideBIn);
};  // Class Triangle

Rectangle::Rectangle(double sideAIn, double sideBIn)
{
    sideA = sideAIn;
    sideB = sideBIn;
    calcPeri();
    calcArea();
    return;
}  // Rectangle constructor

void Rectangle::calcArea()
{
    area = sideA * sideB;
    return;
}  // Rectangle calcArea

void Rectangle::calcPeri()
{
    perimeter = 2 * (sideA + sideB);
    return;
}  // Rectangle calcPeri
Example: Modified Square Class

class Square : public Polygons
{
    private:
        double side;
        virtual void calcArea ();
        virtual void calcPeri ();
    public:
        Square (double sideIn);
}; // Class Square

Square :: Square (double sideIn)
{
    side = sideIn;
    calcPeri();
    calcArea();
    return;
} // Square constructor

void Square :: calcArea ()
{
    area = side * side;
    return;
} // Square calcArea

void Square :: calcPeri ()
{
    perimeter = 4 * side;
    return;
} // Square calcPeri
Example: Modified main()

```cpp
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;

#include "p12-11.h"    // Polygon class
#include "p12-12.h"    // Triangle class
#include "p12-13.h"    // Rectangle class
#include "p12-14.h"    // Square class

int main() {
    cout << "Start Polygon Demonstration\n\n";
    Triangle tri(2, 3, 4);
    tri.printPeri();
    tri.printArea();

    Rectangle rec(3, 4);
    rec.printPeri();
    rec.printArea();

    Square sqr(3);
    sqr.printPeri();
    sqr.printArea();

    cout << "\nEnd Polygon Demonstration\n";
    return 0;
} // main
//============================= End of Program

/* Results
Start Polygon Demonstration
Perimeter of polygon:  9.0
Area of polygon:      2.9
Perimeter of polygon: 14.0
Area of polygon:      12.0
Perimeter of polygon: 12.0
Area of polygon:      9.0
End Polygon Demonstration */
```
Example: Modified main() using Base Class pointer

```cpp
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;

#include "p12-11.h"     // Polygon class
#include "p12-12.h"     // Triangle class
#include "p12-13.h"     // Rectangle class
#include "p12-14.h"     // Square class

int main ()
{
    int a;
    cout << "Start Polygon Demonstration\n\n";
    Polygons* ptrPoly;

    ptrPoly = new Triangle (2, 3, 4);
    ptrPoly->printPeri();
    ptrPoly->printArea();

    ptrPoly = new Rectangle (3, 4);
    ptrPoly->printPeri();
    ptrPoly->printArea();

    ptrPoly = new Square (3);
    ptrPoly->printPeri();
    ptrPoly->printArea();

    cout << "\nEnd Polygon Demonstration\n";
    cin >> a;
    return 0;
} // main
```

/* Results
Start Polygon Demonstration
Perimeter of polygon:  9.0
Area of polygon: 2.9
Perimeter of polygon: 14.0
Area of polygon: 12.0
Perimeter of polygon: 12.0
Area of polygon: 9.0
End Polygon Demonstration */
Review: Casting Standard Data Types

- Can convert a standard data type to another standard data type using either implicit or explicit casting (static casting)

- Can convert a pointer to a standard data type to a pointer to another standard data type using explicit casting (implicit is not allowed)

→ Let’s expand the concept to objects and pointer to objects
Class: Upcasting and Downcasting

- **Upcasting**
  - Casting Derived class to Base class

- **Downcasting**
  - Casting Base class to Derived class

Casting of objects (or pointer to objects) is applicable when the classes form a hierarchy of inheritance.
Upcasting Objects

Note:
A derived class object is a special type of its base class object

```cpp
class Base
{
    ...
}; // Base
class Derived: public Base
{
    ...
}; // Derived
    ...
int main (void)
{
    // Local Definitions
    Base    baseObj;
    Derived derObj;
    ...
    // Statements
    baseObj = derObj;
    ...
} // main
```
Implicit and explicit class conversion

```cpp
int main ()
{
    cout << "Before assign: ";
    Base baseObj (1);
    baseObj.print ();

    Derived derObj (3, 5 );
    baseObj = derObj;  // implicit upcasting
    cout << "After assign: ";
    baseObj.print ();
    return 0;
}  // main

/* Results
Before assign: Value of
baseData in class Base is: 1
After assign: Value of baseData
in class Base is: 3
*/
```

If we use explicit upcasting,
```
baseObj = static_cast<Base>derObj;
```
This has the same effect with the implicit casting.
Downcasting objects

- Downcasting of objects is **not** allowed in C++

- Notice that a derived class object normally has more data member than the base class
  - The language system does not know how to initialize the extra data
Upcasting Pointers to Objects

```
BaseClass* basePtr;

DervClass* derivPtr = new DervClass();

basePtr = static_cast<BaseClass*>(derivPtr);
```
Downcasting Pointers to Objects

- Downcasting pointers to objects is possible only with dynamic casting.

- **Dynamic casting**: casting in which the type is determined during the execution of the program, as opposed to during the compilation.
  - Recall dynamic binding.

- Dynamic casting can be done only with polymorphic classes—that is, classes using virtual functions.

- Syntax:
  ```
  dynamic_cast<Type> (data)
  ```
Downcasting Pointers to Objects (Valid case)

Dynamic casting is valid only when the base class pointer is *already* pointing to a derived class object.
Downcasting Pointers to Objects (Invalid case)

cannot downcast when the base class pointer is not pointing to a derived class object
⇒ the cast returns a null pointer value (0)
Multiple Inheritance

- Derived class inherits from multiple base classes
#include <iostream>
using namespace std;

class BaseOne
{
    private:
        int baseOneData;

    public:
        BaseOne (int baseOneIn);
        void print () const;
}; // BaseOne

BaseOne :: BaseOne (int baseOneIn)
{   cout << "Constructor for BaseOne called " << endl;
    baseOneData = baseOneIn;
} // BaseOne constructor

void BaseOne::print () const
{   cout << baseOneData << endl;
} // BaseOne :: print

class BaseTwo
{
    private:
        int baseTwoData;

    public:
        BaseTwo (int baseTwoIn);
        void print () const;
}; // BaseTwo

BaseTwo :: BaseTwo (int baseTwoIn)
{   cout << "Constructor for BaseTwo called " << endl;
    baseTwoData = baseTwoIn;
} // BaseTwo constructor

void BaseTwo :: print () const
{   cout << baseTwoData << endl;
} // BaseTwo :: print

class MultiInherit: public BaseOne, public BaseTwo
{
    private:
        int multiData;

    public:
        MultiInherit (int baseOneIn, int baseTwoIn,
                      int multiIn);
        void print () const;
}; // MultiInherit

MultiInherit :: MultiInherit (int baseOneIn,
                             int baseTwoIn,
                             int multiIn)
    : BaseOne (baseOneIn), BaseTwo (baseTwoIn)
{   cout << "Constructor for MultiInherit called" << endl;
    multiData = multiIn;
} // MultiInherit constructor

void MultiInherit :: print () const
{   BaseOne :: print ();
    BaseTwo :: print ();
    cout << multiData << endl;
} // MultiInherit :: print

int main ()
{   MultiInherit multi (1, 2, 12);
    multi.print ();
    return 0;
} // main

/* Results
Constructor for BaseOne called
Constructor for BaseTwo called
Constructor for MultiInherit called
1
2
12 */
Another Example: C++ File Library
Aggregation

- Inclusion of objects within an object
- Has-a relationship

(a) Derived is a Base

(b) Class aggregates *two classes*
Example: Aggregation
Instantiating an aggregated object

class SalaryEmployee : Employee
{
    ...
} // SalaryEmployee

// ======== Constructor ========
SalaryEmployee :: SalaryEmployee
    (int idIn, float salaryIn)
    : Employee (idIn)
{
    ...
} // SalaryEmployee

class Student
{
    private:
        Date dob;
    ...
} // Student

// ======== Constructor ========
class Student :: Student
    (long id, int units, long grPts,
     unsigned short day, 
     unsigned short mon, 
     unsigned short year)
    : dob (day, mon, year)
{
    ...
}
Example: Implementation of Aggregation

class Student
{
    private:
        long id;
        int units;
        int grPts;
        Date dob;

    public:
        Student (long idIn, int unitsIn, int grPtsIn,
            unsigned short dayIn,
            unsigned short monIn,
            unsigned short yearIn)
            : dob (dayIn, monIn, yearIn)
        {
            id = idIn;
            units = unitsIn;
            grPts = grPtsIn;
        } // Constructor

        void print () const
        {
            cout << "Student: " << id << " Units: " << units
                << "nGr Pts: " << setw (4) << grPts
                << " Birth Date: ";
            dob.print();
            cout << endl;
        } // Student

}; // Class Student
Questions?