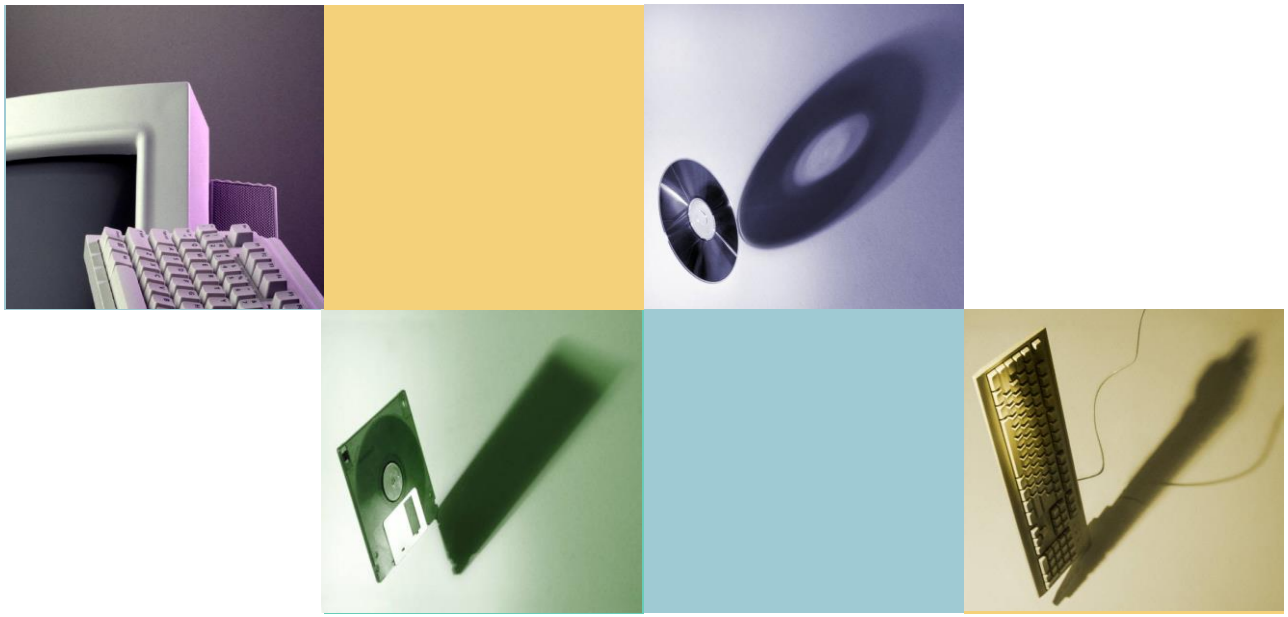
Object-Oriented Programming



Chuan-Kang Ting

Dept of Computer Science and Information Engineering

National Chung Cheng University

Chapter 14

Inheritance



Introduction

- **Object-Oriented Programming**
  - Popular and powerful programming technique
  - Recall: **PIE**
  - Provides abstraction dimension called *inheritance*
- **Inheritance?**
  - First, a very general form of class is defined
  - Specialized versions then
    - Inherit properties of general class
    - Add to it or/and modify its functionality for appropriate use



Inheritance

- **New class inherited from another class**
  - **Base** class
    - **General** class from which others derive
  - **Derived** class
    - New class
    - Automatically has ALL the base class's:
      - Member variables
      - Member functions
    - Can have **additional** member functions and variables



Example of Inheritance

- **Example – class Employee**
  - Hierarchy:

    Employees → ***General***

    - Salaried employees
    - Hourly employees

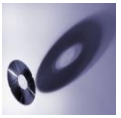
    } ***Specific*** *subset of employee*



Base Classes (1)

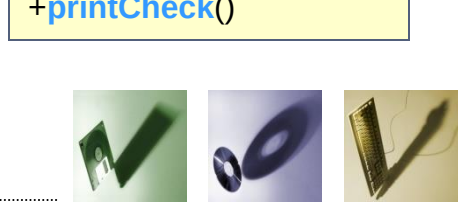
- **General aspect – Employee**
  - General concept is helpful (for what?)
  - All employees have
    - Names
    - Social security numbers (ssn)
    - Same member functions for setting and changing data
      - Accessor functions
      - Mutator functions
  - So "general" class can contain all these "things" about employees
  - However, we won't have objects of this class
    - Since no one is just an employee

Employee
-names: string -ssn: string -netPay: double
+setName(string) +setSsn(string) +setNetPay(double) +getName(): string +getSsn(): string +getNetPay(): double +printCheck()


Base Classes (2)

- **General aspect – Employee** (cont'd)
  - Define derived class for different kinds of employees
  - Consider `printCheck()` function:
    - Will always be "redefined" in derived classes
      - So *different* employee types can have *different* checks
    - Makes no sense really for "undifferentiated" employee
    - Thus, `printCheck()` is implemented in Employee class by just saying
      - Error message: "printCheck called for undifferentiated employee!! Aborting…"

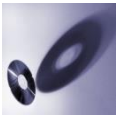


Base Classes (3)

- **Code for Employee**

Display 14.1 **Interface for the Base Class** Employee

```
1
2   //This is the header file employee.h.
3   //This is the interface for the class Employee.
4   //This is primarily intended to be used as a base class to derive
5   //classes for different kinds of employees.
6   #ifndef EMPLOYEE_H
7   #define EMPLOYEE_H

8   #include <string>
9   using std::string;

10  namespace SavitchEmployees
11  {

12      class Employee
13      {
14      public:
15          Employee( );
16          Employee(string theName, string theSsn);
17          string getName( ) const;
18          string getSsn( ) const;
19          double getNetPay( ) const;
20          void setName(string newName);
21          void setSsn(string newSsn);
22          void setNetPay(double newNetPay);
23          void printCheck( ) const;
24      private:
25          string name;
26          string ssn;
27          double netPay;
28      };

29  }//SavitchEmployees

30  #endif //EMPLOYEE_H
```


Derived Classes

- **Derived classes from Employee class**
  - Automatically have
    - ALL member variables
    - ALL member functions

    We say: the derived class inherits the member variables and member functions
  - Can do more:
    - **Redefine** existing members
    - **Add** new members



Example – HourlyEmployee

```
2   //This is the header file hourlyemployee.h.
3   //This is the interface for the class HourlyEmployee.
4   #ifndef HOURLYEMPLOYEE_H
5   #define HOURLYEMPLOYEE_H

6   #include <string>
7   #include "employee.h"

8   using std::string;

9   namespace SavitchEmployees
10  {
11      class HourlyEmployee : public Employee
12      {
13      public:
14          HourlyEmployee( );
15          HourlyEmployee(string theName, string theSsn,
16                         double theWageRate, double theHours);
17          void setRate(double newWageRate);
18          double getRate( ) const;
19          void setHours(double hoursWorked);
20          double getHours( ) const;
21          void printCheck( );
22      private:
23          double wageRate;
24          double hours;
25      };

26  }//SavitchEmployees

27  #endif //HOURLYEMPLOYEE_H
```

Definition begins same as others

- #ifndef structure
- includes required libraries
- also includes employee.h

Inheritance

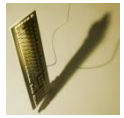
- specifies ***publicly*** **inherited** from Employee

You only list the declaration of an inherited member function if you want to change the definition of the function.

Add

Redefine

Add

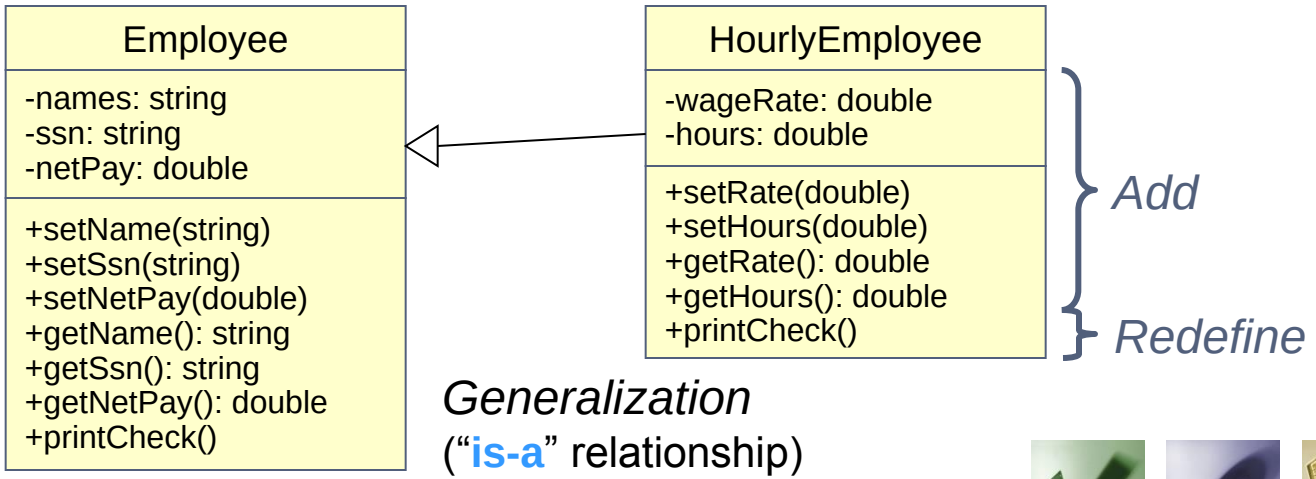


Notes about HourlyEmployee (1)

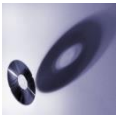
- **Inheritance**

  class ***Derived***_Class_Name **:** **public** ***Base***_Class_Name

  – Derived class interface only lists new or to be redefined members
    - Since all others inherited are already defined
    - That is, “all” employees have ssn, name, etc

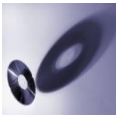


Generalization
(“**is-a**” relationship)


Notes about HourlyEmployee (2)

- **Derived class HourlyEmployee**
  - Interface only lists new or to be redefined members
  - New:
    - Constructors
    - Member variables: wageRate, hours
    - Member functions: setRate(), getRate(), setHours(), getHours()
  - Redefine:
    - printCheck()
      - Specialized to hourly employees
      - Its definition must be in HourlyEmployee class's implementation

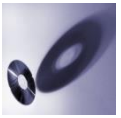

Notes about HourlyEmployee (3)

- **Objects of derived class have more than one type**
  - Are both types of base class and derived class
    - An hourly employee is an employee
    - NOT vice versa!
      - After all, an employee is not necessarily an hourly employee

  → Every object of class HourlyEmployee can be used *anyplace* an object of class Employee can be used

    - A convenient way to identify the type in practice

```
void fireEmployee(Employee& badLuck);

int main()
{
  HourlyEmployee a;
  SalarizedEmployee b;
  ...
  fireEmployee(a);
  fireEmployee(b);
}
```


Inheritance Terminology

- **Family relationship**
  - Parent class
    - Refers to base class
  - Child class
    - Refers to derived class
  - Ancestor class
    - Class that is a parent of a parent …
  - Descendant class
    - Opposite of ancestor



Constructors in Derived Classes (1)

- **Constructor in base class is NOT inherited in derived classes!**
  - But it can be invoked within derived class constructor
    - Which is all we need
  - Derived class needs to initialize all member variables:
    - Those **inherited** from base class
      - Using base class constructor
      - Done first
    - Those **added** by derived class
      - Using derived class constructor
    - Inheritance: A←B←C
      - Invoke constructors in sequence: A, B, C



Constructors in Derived Classes (2)

- **Example**
  - HourlyEmployee **constructor**

```
HourlyEmployee::HourlyEmployee(string theName,
       string theNumber, double theWageRate, double theHours)
     : Employee(theName, theNumber),
                wageRate(theWageRate), hours(theHours)
{
     //Deliberately empty
}
```

Base class constructor

Initialization section

  - HourlyEmployee **default constructor**

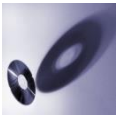
Base class constructor

```
HourlyEmployee::HourlyEmployee( ) : Employee(),
                                    wageRate(0),hours(0)
{
     //Deliberately empty
}
```

Works fine if omitted

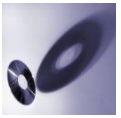
Once derived class constructor does not invoke base class constructor, the default base class constructor will be invoked **automatically**

Should always invoke base class constructor!

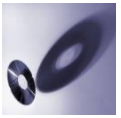

Pitfall: Base Class Private Members (1)

- **How private?**
  - Private member in a base class is not accessible for *any* other class, NOT even for *derived class*
    - Derived class cannot directly access private member
    - Although derived class “inherits” private members in base class
  - **Why?**


Pitfall: Base Class Private Members (2)

- **Private variable**
  - Still can be accessed *indirectly* via accessor or mutator member function
- **Private member function**
  - Is simply NOT available
  - Just as if it was not inherited
    - Reasons:
      - Private member functions should be simply helping functions
      - Their use must be limited to the class in which they are defined

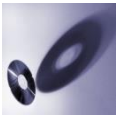




The `protected` Qualifier

- **New classification of class members**
- **Protected members**
  - As if *private* in any class other than derived class
  - Can be accessed by name in a derived class
    - Also accessible for all descendant classes

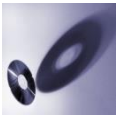
  → Not as open as *public*;
  not as closed as *private*
    - Many programming authorities feel this violates information hiding
      - All member variables should be private
      - You make your own decision on whether to use it!

```
class Employee
{
  public:
    ...
  protected:
    string name;
    string ssn;
    double netPay;
};
```


Redefining Member Functions (1)

- **Recall interface of derived class:**
  - Contains declarations for new member functions
  - Also contains declarations for inherited member functions to be changed
    - **Must list**, even though the same as in the base class
    - Inherited member functions are inherited *unchanged* if they are NOT declared in the interface of derived class
- **Implementation of derived class will**
  - Define **new** member functions
  - **Redefine** inherited functions as declared

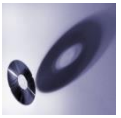




Redefining Member Functions (2)

- **Redefining vs. overloading**
  - Very different!
  - **Redefining** in derived class
    - SAME parameter list (including number and types)
    - Essentially re-writes same function
  - **Overloading**
    - Different parameter list
    - Define new function that takes different parameters
    - Overloaded functions must have different signatures

Signature

A function's signature is the function's **name** with the **sequence of types** in the parameter list,
NOT including `const`, ampersand `&`, and return type

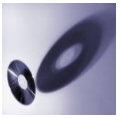

Redefining Member Functions (3)

- **Access to a 'redefined' base function**
  - Base class member function is "not **lost**" after being redefined
  - Use scope resolution operator
  - e.g.

```
HourlyEmployee Sally;

Sally.printCheck();
Sally.Employee::printCheck();
```

invoke printCheck() redefined in HourlyEmployee

invoke the original printCheck() in Employee

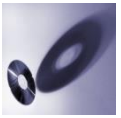

Functions Not Inherited

- **In general**
  - All “normal” functions in base class are inherited in derived class
- **Exception**
  - Constructors (we’ve seen)
  - Destructors
  - Copy constructor
  - Assignment operator =

The Big Three

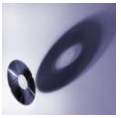
for any class that uses **pointers** and the **new** operator, it is safest to define your own *copy constructor*, *overloaded* =, and *destructor*

→ **need Deep Copy**


Assignment Operators & Copy Constructors

- **Both are NOT inherited**
- **Two versions**
  - Default
    - Generated automatically if not provided
    - Shallow copy
  - Redefined / Overloaded
    - Needed when class member variables involve pointers, dynamic arrays, or other dynamic data
    - Deep copy


(1) Assignment Operator in Derived Classes

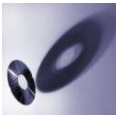
- **Example**
  - **`Derived`** is a class derived from **`Base`**

```
Derived& Derived::operator =(const Derived& rSide)
{
  Base::operator =(rSide);
  ...
}
```

    - Firstly, a call to assignment operator of Base class
      - This takes care of all inherited member variables
      - You must have a correctly functioning assignment operator for Base
    - Then, set the new member variables of Derived class

  → **Cooperation** of Base and Derived classes

    - Happens also in copy constructor and destructor

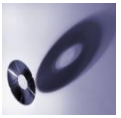




(2) Copy Constructor in Derived Classes

- **Example**
  - **`Derived`** is a class derived from **`Base`**

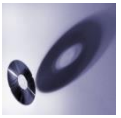
```
Derived::Derived(const Derived& Object)
                    :Base(Object),...
{
}
```

    - Firstly, invoke copy constructor for Base class
      - Sets inherited member variables
      - You must have a correctly functioning copy constructor for Base
      - Note that `Object` is of type **Derived** as well as type **Base**; Therefore, `Object` is a *legal* argument to copy constructor for Base class


(3) Destructors in Derived Classes

- **When derived class destructor is invoked:**
  - Automatically calls base class destructor
    - Since base class is being out of scope as well
    - So no need for explicit call
  - Cooperation
    - Base and Derived destructors **`delete`** their own variables respectively
  - Sequence
    - Given A←B←C
    - Invoke destructors of C, B, and finally A
      - Opposite of how constructors are called


“Is a” versus “Has a”

- **“Is a”**
  - Inheritance is considered as an “is a" relationship between classes (generalization)
  - e.g. An HourlyEmployee **is an** Employee
    A Coupe **is an** Automobile
- **“Has a”**
  - A class contains objects of another class as its member data (association)
  - e.g. An AirPlane **has a** JetEngine

To distinguish, just follow what sounds most natural in English

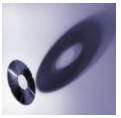


Protected and Private Inheritance (1)

- **Protected / Private inheritance**
  - New inheritance form
  - Changes the access authority of members from base class
  - Syntax

```
class SalariedEmployee : protected Employee
```

    - Members that are *public in the base class* are then ***protected*** in the derived class
  - With protected/private inheritance, an object of derived class is NOT an object of base class
    - The “is a” relationship does NOT hold in this inheritance
    - Therefore, rarely used

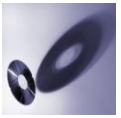




Protected and Private Inheritance (2)

- **Change in access authority**

Access specifier in Base class	Type of inheritance		
	public	**protected**	**private**
public	Public	Protected	Private
protected	Protected	Protected	Private
private	Private	Private	Private

Private: can NOT be accessed by name in the derived class


Multiple Inheritance

- **Derived class can have more than one base class**
  - Syntax
    - Separate base classes by commas

```
class DerivedM : public Base1, Base2, ...
```

  - Numerous possibilities for ambiguity
    - What if two base classes have a function/variable with the same name and parameter types? Which is inherited?
  - Some authorities consider it so dangerous that should NOT be used at all
    - Only used by experienced programmers!



Summary (1)

- **Inheritance**
  - Provides a tool for code reuse
  - Considered as “is a” relationship
- **Derived class**
  - Inherits the members of base class
    - Can redefine member functions inherited from base class
    - Cannot access private members in base class
      - Private member function effectively are not inherited
  - Can add members



Summary (2)

- **Derived class** (cont'd)
  - Can access protected members
  - Not inherit
    - Constructors
    - Destructors
    - Overloaded assignment operator
    - Copy constructor
  - Protected / private inheritance
  - Multiple inheritance