



Chapter 14

Inheritance



Learning Objectives

§ Inheritance Basics

- § Derived classes, with constructors

- § protected: qualifier

- § Redefining member functions

- § Non-inherited functions

§ Programming with Inheritance

- § Assignment operators and copy constructors

- § Destructors in derived classes

- § Multiple inheritance



Introduction to Inheritance

- § Object-oriented programming
 - § Powerful programming technique
 - § Provides abstraction dimension called *inheritance*
- § General form of class is defined
 - § Specialized versions then inherit properties of general class
 - § And add to it/modify it's functionality for it's appropriate use



Inheritance Basics

- § New class inherited from another class
- § Base class
 - § 'General' class from which others derive
- § Derived class
 - § New class
 - § Automatically has base class's:
 - § Member variables
 - § Member functions
 - § Can then add additional member functions and variables



Derived Classes

- § Consider example:
Class of 'Employees'
- § Composed of:
 - § Salaried employees
 - § Hourly employees
- § Each is 'subset' of employees
 - § Another might be those paid fixed wage each month or week



Derived Classes

- § Don't 'need' type of generic 'employee'
- § Since no one's just an 'employee'
- § General concept of employee helpful!
- § All have names
- § All have social security numbers
- § Associated functions for these 'basics' are same among all employees
- § So 'general' class can contain all these 'things' about employees



Employee Class

- § Many members of 'employee' class apply to all types of employees
 - § Accessor functions
 - § Mutator functions
 - § Most data items:
 - § SSN
 - § Name
 - § Pay
- § We won't have 'objects' of this class, however



Employee Class

- § 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
 - § So function printCheck() in Employee class says just that
 - § Error message stating "printCheck called for undifferentiated employee!! Aborting..."



Deriving from Employee Class

- § Derived classes from Employee class:
 - § Automatically have all member variables
 - § Automatically have all member functions
- § Derived class said to 'inherit' members from base class
- § Can then redefine existing members and/or add new members



Interface for HourlyEmployee Class

Display 14.3,
page 588

Display 14.3 Interface for the Derived Class HourlyEmployee

```
1
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
```

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



HourlyEmployee Class Interface

- § Note definition begins same as any other
- § `#ifndef` structure
- § Includes required libraries
- § Also includes `employee.h!`
- § And, the heading:

```
class HourlyEmployee : public Employee  
{ ...
```
- § Specifies 'publicly inherited' from `Employee` class



HourlyEmployee Class Additions

- § Derived class interface only lists new or 'to be redefined' members
 - § Since all others inherited are already defined
 - § i.e.: 'all' employees have ssn, name, etc.
- § HourlyEmployee adds:
 - § Constructors
 - § wageRate, hours member variables
 - § setRate(), getRate(), setHours(), getHours()
member functions



HourlyEmployee Class Redefinitions

- § HourlyEmployee redefines:
 - § printCheck() member function
 - § This 'overrides' the printCheck() function implementation from Employee class
- § It's definition must be in HourlyEmployee class's implementation
 - § As do other member functions declared in HourlyEmployee's interface
 - § New and 'to be redefined'



Inheritance Terminology

- § Common to simulate family relationships
- § Parent class
 - § Refers to base class
- § Child class
 - § Refers to derived class
- § Ancestor class
 - § Class that's a parent of a parent ...
- § Descendant class
 - § Opposite of ancestor



Constructors in Derived Classes

- § Base class constructors are NOT inherited in derived classes!
- § But they can be invoked within derived class constructor
 - § Which is all we need!
- § Base class constructor must initialize all base class member variables
 - § Those inherited by derived class
 - § So derived class constructor simply calls it
 - § 'First' thing derived class constructor does



Derived Class Constructor Example

§ Consider syntax for HourlyEmployee constructor:

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

§ Portion after : is 'initialization section'

§ Includes invocation of Employee constructor



Another HourlyEmployee Constructor

§ A second constructor:

```
HourlyEmployee::HourlyEmployee()  
    : Employee(),      wageRate(0),  
                          hours(0)
```

```
{  
    //Deliberately empty  
}
```

§ Default version of base class constructor is called (no arguments)

§ Should always invoke one of the base class's constructors



Constructor: No Base Class Call

- § Derived class constructor should always invoke one of the base class's constructors
- § If you do not:
 - § Default base class constructor automatically called
- § Equivalent constructor definition:

```
HourlyEmployee::HourlyEmployee()  
    : wageRate(0), hours(0)  
{ }
```



Pitfall: Base Class Private Data

- § Derived class 'inherits' private member variables
 - § But still cannot directly access them
 - § Not even through derived class member functions!
- § Private member variables can ONLY be accessed 'by name' in member functions of the class they're defined in



Pitfall: Base Class Private Member Functions

- § Same holds for base class member functions
- § Cannot be accessed outside interface and implementation of base class
- § Not even in derived class member function definitions



Pitfall: Base Class Private Member Functions Impact

- § Larger impact here vs. member variables
- § Member variables can be accessed indirectly via accessor or mutator member functions
- § Member functions simply not available
- § This is 'reasonable'
- § Private member functions should be simply 'helper' functions
- § Should be used only in class they're defined



The protected: Qualifier

- § New classification of class members
- § Allows access 'by name' in derived class
 - § But nowhere else
 - § Still no access 'by name' in other classes
- § In class it's defined à acts like private
- § Considered 'protected' in derived class
 - § To allow future derivations
- § Many feel this 'violates' information hiding



Redefinition of Member Functions

- § Recall interface of derived class:
 - § Contains declarations for new member functions
 - § Also contains declarations for inherited member functions to be changed
 - § Inherited member functions NOT declared:
 - § Automatically inherited unchanged
- § Implementation of derived class will:
 - § Define new member functions
 - § Redefine inherited functions as declared



Redefining vs. Overloading

- § Very different!
- § Redefining in derived class:
 - § SAME parameter list
 - § Essentially 're-writes' same function
- § Overloading:
 - § Different parameter list
 - § Defined 'new' function that takes different parameters
 - § Overloaded functions must have different signatures



A Function's Signature

- § Recall definition of a 'signature':
 - § Function's name
 - § Sequence of types in parameter list
 - § Including order, number, types
- § Signature does NOT include:
 - § Return type
 - § const keyword
 - § &



Accessing Redefined Base Function

§ When redefined in derived class, base class's definition not 'lost'

§ Can specify it's use:

Employee JaneE;

HourlyEmployee SallyH;

JaneE.printCheck(); → calls Employee's
printCheck function

SallyH.printCheck(); → calls HourlyEmployee
printCheck function

SallyH.Employee::printCheck(); → Calls Employee's
printCheck function!

§ Not typical here, but useful sometimes



Functions Not Inherited

- § All 'normal' functions in base class are inherited in derived class
- § Exceptions:
 - § Constructors (we've seen)
 - § Destructors
 - § Copy constructor
 - § But if not defined, generates 'default' one
 - § Recall need to define one for pointers!
 - § Assignment operator
 - § If not defined à default



Assignment Operators and Copy Constructors

- § Recall: overloaded assignment operators and copy constructors NOT inherited
- § But can be used in derived class definitions
- § Typically MUST be used!
- § Similar to how derived class constructor invokes base class constructor



Assignment Operator Example

- § Given 'Derived' is derived from 'Base':
Derived& Derived::operator =(const Derived & rightSide)
{
 Base::operator =(rightSide);
 ...
}
- § Notice code line
 - § Calls assignment operator from base class
 - § This takes care of all inherited member variables
 - § Would then set new variables from derived class...



Copy Constructor Example

- § Consider:
Derived::Derived(const Derived& Object)
 : Base(Object), ...

{...}
- § After : is invocation of base copy constructor
- § Sets inherited member variables of derived class object being created
- § Note Object is of type Derived; but it's also of type Base, so argument is valid



Destructors in Derived Classes

- § If base class destructor functions correctly
- § Easy to write derived class destructor
- § When derived class destructor is invoked:
 - § Automatically calls base class destructor!
 - § So no need for explicit call
- § So derived class destructors need only be concerned with derived class variables
 - § And any data they 'point' to
 - § Base class destructor handles inherited data automatically



Destructor Calling Order

- § Consider:
 - class B derives from class A
 - class C derives from class B
- A \supset B \supset C
- § When object of class C goes out of scope:
 - § Class C destructor called 1st
 - § Then class B destructor called
 - § Finally class A destructor is called
- § Opposite of how constructors are called



“Is a” vs. “Has a” Relationships

§ Inheritance

§ Considered an “Is a” class relationship

§ e.g.: An HourlyEmployee “is a” Employee

§ A Convertible “is a” Automobile

§ A class contains objects of another class as it’s member data

§ Considered a “Has a” class relationship

§ e.g.: One class “has a” object of another class as it’s data



Protected and Private Inheritance

§ New inheritance 'forms'

§ Both are rarely used

§ Protected inheritance:

```
class SalariedEmployee : protected Employee  
{...}
```

§ Public members in base class become protected in derived class

§ Private inheritance:

```
class SalariedEmployee : private Employee  
{...}
```

§ All members in base class become private in derived class



Multiple Inheritance

- § Derived class can have more than one base class!
- § Syntax just includes all base classes separated by commas:
class derivedMulti : public base1, base2
{...}
- § Possibilities for ambiguity are endless!
- § Dangerous undertaking!
 - § Some believe should never be used
 - § Certainly should only be used by experienced programmers!



Summary 1

- § Inheritance provides code reuse
- § Allows one class to 'derive' from another, adding features
- § Derived class objects inherit members of base class
- § And may add members
- § Private member variables in base class cannot be accessed 'by name' in derived
- § Private member functions are not inherited



Summary 2

- § Can redefine inherited member functions
 - § To perform differently in derived class
- § Protected members in base class:
 - § Can be accessed 'by name' in derived class member functions
- § Overloaded assignment operator not inherited
 - § But can be invoked from derived class
- § Constructors are not inherited
 - § Are invoked from derived class's constructor