2023

Program Design II

Kun-Ta Chuang
Department of Computer Science and Information Engineering
National Cheng Kung University

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

Display 14.3 Interface for the Derived Class HourlyEmployee (1 of 2)

Display 14.3 Interface for the Derived Class HourlyEmployee

```

//This is the header file hourlyemployee.h.
//This is the interface for the class HourlyEmployee.
#ifndef HOURLYEMPLOYEE_H
#define HOURLYEMPLOYEE_H

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

using std::string;

namespace SavitchEmployees
{
```

Display 14.3 Interface for the Derived Class HourlyEmployee (2 of 2)

```
    class HourlyEmployee : public Employee
    {
    public:
        HourlyEmployee( );
        HourlyEmployee(string theName, string theSsn,
                                 double theWageRate, double theHours);
        void setRate(double newWageRate);
        double getRate( ) const;
        void setHours(double hoursWorked);
        double getHours( ) const;
        void printCheck( ) ;
    private:
        double wageRate;
        double hours;
    };

}//SavitchEmployees

#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 → 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 → 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 ← B ← 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 be 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

Learning Objectives

- Virtual Function Basics
  - Late binding
  - Implementing virtual functions
  - When to use a virtual function
  - Abstract classes and pure virtual functions
- Pointers and Virtual Functions
  - Extended type compatibility
  - Downcasting and upcasting
  - C++ "under the hood" with virtual functions

Virtual Function Basics

- Polymorphism
  - Associating many meanings to one function
  - Virtual functions provide this capability
  - Fundamental principle of object-oriented programming!
- Virtual
  - Existing in "essence" though not in fact
- Virtual Function
  - Can be "used" before it’s "defined"

Figures Example

- Best explained by example:
- Classes for several kinds of figures
  - Rectangles, circles, ovals, etc.
  - Each figure an object of different class
    - Rectangle data: height, width, center point
    - Circle data: center point, radius
- All derive from one parent-class: Figure
- Require function: draw()
  - Different instructions for each figure

Figures Example 2

- Each class needs different *draw* function
- Can be called "draw" in each class, so:

```
Rectangle r;
Circle c;
r.draw();  //Calls Rectangle class's draw
c.draw(); //Calls Circle class's draw
```

- Nothing new here yet…

Figures Example: center()

- Parent class Figure contains functions that apply to "all" figures; consider: center(): moves a figure to center of screen
  - Erases 1st, then re-draws
  - So Figure::center() would use function draw() to re-draw
  - Complications!
    - Which draw() function?
    - From which class?

Figures Example: New Figure

- Consider new kind of figure comes along:
  Triangle class
  derived from Figure class
- Function center() inherited from Figure
  - Will it work for triangles?
  - It uses draw(), which is different for each figure!
  - It will use Figure::draw() → won't work for triangles
- Want inherited function center() to use function Triangle::draw() NOT function Figure::draw()
  - But class Triangle wasn't even WRITTEN when Figure::center() was! Doesn't know "triangles"!

Figures Example: Virtual!

- Virtual functions are the answer
- Tells compiler:
  - "Don’t know how function is implemented"
  - "Wait until used in program"
  - "Then get implementation from object instance"
- Called late binding or dynamic binding
  - Virtual functions implement late binding

Virtual Functions: Another Example

- Bigger example best to demonstrate
- Record-keeping program for automotive parts store
  - Track sales
  - Don't know all sales yet
  - 1st only regular retail sales
  - Later: Discount sales, mail-order, etc.
    - Depend on other factors besides just price, tax

Virtual Functions: Auto Parts

- Program must:
  - Compute daily gross sales
  - Calculate largest/smallest sales of day
  - Perhaps average sale for day
- All come from individual bills
  - But many functions for computing bills will be added "later"!
    - When different types of sales added!
- So function for "computing a bill" will be virtual!

Class Sale Definition

- class Sale

```
{
public:
      Sale();
      Sale(double thePrice);
      double getPrice() const;
      virtual double bill() const;
      double savings(const Sale& other) const;
private:
      double price;
};
```

Member Functions savings and operator <

- double Sale::savings(const Sale& other) const
```
{
    return (bill() – other.bill());
}
```
- bool operator < (const Sale& first, const Sale& second)
```
{
    return (first.bill() < second.bill());
}
```
- Notice BOTH use member function bill()!

Class Sale

- Represents sales of single item with no added discounts or charges.
- Notice reserved word "virtual" in declaration of member function *bill*
  - Impact: Later, derived classes of Sale can define THEIR versions of function bill
  - Other member functions of Sale will use version based on object of derived class!
  - They won't automatically use Sale's version!

Derived Class DiscountSale Defined

- ```
  class DiscountSale   :  public Sale
  {
  public:
          DiscountSale();
          DiscountSale(        double thePrice,
                               double the Discount);
          double getDiscount() const;
          void setDiscount(double newDiscount);
          double bill() const;
  private:
          double discount;
  };
  ```
  ```

DiscountSale's Implementation of bill()

- double DiscountSale::bill() const

```
{
    double fraction = discount/100;
    return (1 – fraction)*getPrice();
}
```

- Qualifier "virtual" does not go in actual function definition
  - "Automatically" virtual in derived class
  - Declaration (in interface) not required to have "virtual" keyword either (but usually does)

DiscountSale's Implementation of bill()

- Virtual function in base class:
  - "Automatically" virtual in derived class
- Derived class declaration (in interface)
  - Not required to have "virtual" keyword
  - But typically included anyway, for readability

Derived Class DiscountSale

- DiscountSale's member function bill() implemented differently than Sale's
  - Particular to "discounts"
- Member functions *savings* and "<"
  - Will use this definition of bill() for all objects of DiscountSale class!
  - Instead of "defaulting" to version defined in Sales class!

Virtual: Wow!

- Recall class Sale written long before derived class DiscountSale
  - Members savings and "<" compiled before even had ideas of a DiscountSale class
- Yet in a call like:
  DiscountSale d1, d2;
  d1.savings(d2);
  - Call in savings() to function bill() knows to use definition of bill() from DiscountSale class
- Powerful!

Virtual: How?

- To write C++ programs:
  - Assume it happens by "magic"!
- But explanation involves late binding
  - Virtual functions implement late binding
  - Tells compiler to "wait" until function is used in program
  - Decide which definition to use based on calling object
- Very important OOP principle!

Overriding

- Virtual function definition changed in a derived class
  - We say it's been "overidden"
- Similar to redefined
  - Recall: for standard functions
- So:
  - Virtual functions changed: ***overridden***
  - Non-virtual functions changed: ***redefined***

C++11 **override** keyword

- C++11 includes the **override** keyword to make it clear if a function is overridden or redefined

```
class Sale
{
  public:
    …
    virtual double bill() const;
    …
};

class DiscountSale : public Sale
{
  public:
    …
    double bill() const override;
    …
};
```

Makes it explicit that this function overrides **bill()** in the Sale class

C++11 **final** keyword

- C++11 includes the **final** keyword to prevent a function from being overridden. Useful if a function is overridden but don't want a derived classes to override it again.

```
class Sale
{
  public:
   …
   virtual double bill() const
final;
   …
};
class DiscountSale : public Sale
{
  public:
   …
   double bill() const;
   …
};
```

Cannot override

Results in compiler error

Virtual Functions: Why Not All?

- Clear advantages to virtual functions as we've seen
- One major disadvantage: overhead!
  - Uses more storage
  - Late binding is "on the fly", so programs run slower
- So if virtual functions not needed, should not be used

Pure Virtual Functions

- Base class might not have "meaningful" definition for some of it's members!
  - It's purpose solely for others to derive from
- Recall class Figure
  - All figures are objects of derived classes
    - Rectangles, circles, triangles, etc.
  - Class Figure has no idea how to draw!
- Make it a pure virtual function:
  virtual void draw() = 0;

Abstract Base Classes

- Pure virtual functions require no definition
  - Forces all derived classes to define "their own" version
- Class with one or more pure virtual functions is: abstract base class
  - Can only be used as base class
  - No objects can ever be created from it
    - Since it doesn’t have complete "definitions" of all it’s members!
- If derived class fails to define all pure’s:
  - It’s an abstract base class too

Extended Type Compatibility

- Given:
  Derived is derived class of Base
  - Derived objects can be assigned to objects
    of type Base
  - But NOT the other way!
- Consider previous example:
  - A DiscountSale "is a" Sale, but reverse not true

Extended Type Compatibility Example

- class Pet
  {
  public:

  string name;
  virtual void print() const;

  };
  class Dog : public Pet
  {
  public:

  string breed;
  virtual void print() const;

  };

```
class Pet
{
public:
        string name;
        virtual void print() const;
};
class Dog : public Pet
{
public:
        string breed;
        virtual void print() const;
};
```

Classes Pet and Dog

- Now given declarations:
  Dog vdog;
  Pet vpet;
- Notice member variables name and breed are public!
  - For example purposes only! Not typical!

Using Classes Pet and Dog

- Anything that "is a" dog "is a" pet:
  - vdog.name = "Tiny";
    vdog.breed = "Great Dane";
    vpet = vdog;
  - These are allowable
- Can assign values to parent-types, but not reverse
  - A pet "is not a" dog (not necessarily)

Slicing Problem

- Notice value assigned to vpet "loses" it’s breed field!
  - cout << vpet.breed;
    - Produces ERROR msg!
  - Called slicing problem
- Might seem appropriate
  - Dog was moved to Pet variable, so it should be treated like a Pet
    - And therefore not have "dog" properties
  - Makes for interesting philosphical debate

Slicing Problem Fix

- In C++, slicing problem is nuisance
  - It still "is a" Great Dane named Tiny
  - We’d like to refer to it’s breed even if it’s been treated as a Pet
- Can do so with pointers to dynamic variables

Slicing Problem Example

- Pet *ppet;
  Dog *pdog;
  pdog = new Dog;
  pdog->name = "Tiny";
  pdog->breed = "Great Dane";
  ppet = pdog;
- Cannot access breed field of object pointed to by ppet:
  cout << ppet->breed;   //ILLEGAL!

Slicing Problem Example

- Must use virtual member function:
  ppet->print();
  - Calls print member function in Dog class!
    - Because it's virtual
  - C++ "waits" to see what object pointer ppet is actually pointing to before "binding" call

Virtual Destructors

- Recall: destructors needed to de-allocate dynamically allocated data
- Consider:
  Base *pBase = new Derived;
  …
  delete pBase;
  - Would call base class destructor even though pointing to Derived class object!
  - Making destructor ***virtual*** fixes this!
- Good policy for all destructors to be virtual

Casting

- Consider:
  Pet vpet;
  Dog vdog;
  …
  vdog = static_cast<Dog>(vpet); //ILLEGAL!
- Can't cast a pet to be a dog, but:
  vpet = vdog; // Legal!
  vpet = static_cast<Pet>(vdog); //Also legal!
- Upcasting is OK
  – From descendant type to ancestor type

Downcasting

- Downcasting dangerous!
  - Casting from ancestor type to descended type
  - Assumes information is "added"
  - Can be done with dynamic_cast:

    ```
    Pet *ppet;
    ppet = new Dog;
    Dog *pdog = dynamic_cast<Dog*>(ppet);
    ```

    - Legal, but dangerous!
- Downcasting rarely done due to pitfalls
  - Must track all information to be added
  - All member functions must be virtual

Inner Workings of Virtual Functions

- Don’t need to know how to use it!
  - Principle of information hiding
- Virtual function table
  - Compiler creates it
  - Has pointers for each virtual member function
  - Points to location of correct code for that function
- Objects of such classes also have pointer
  - Points to virtual function table

Summary 1

- Late binding delays decision of which member function is called until runtime
  – In C++, virtual functions use late binding
- Pure virtual functions have no definition
  – Classes with at least one are abstract
  – No objects can be created from abstract class
  – Used strictly as base for others to derive

Summary 2

- Derived class objects can be assigned to base class objects
  - Base class members are lost; slicing problem
- Pointer assignments and dynamic objects
  - Allow "fix" to slicing problem
- Make all destructors virtual
  - Good programming practice
  - Ensures memory correctly de-allocated