Inheritance (with C++)

Starting to cover Savitch Chap. 15

More OS topics in later weeks
(memory concepts, libraries)
Inheritance Basics

- A new class is inherited from an existing class
- Existing class is termed the base class
  - It is the "general" class (a.k.a. superclass, or parent)
- New class is termed the derived class
  - It is the "specific" class (a.k.a. subclass, or child)
  - Automatically has (i.e., "inherits") all of the base class's member functions and variables
  - Can define additional member functions and variables
    - And override inherited virtual functions (but that's a later topic)
Inheritance begets hierarchies

- "Is a" relationships
- Imagine:
  class Basketball is derived from class Ball
- Then:
  any Basketball is a Ball
- Reverse not always true: a Ball can be a Football, or a Baseball, or …
class Employee {
public:
    Employee( );
    Employee(string theName, string theSsn);
    string getName( ) const;
    string getSsn( ) const;
    double getNetPay( ) const;
    void setName(string newName);
    void setSsn(string newSsn);
    void setNetPay(double newNetPay);
    void printCheck( ) const;
private:
    string name;
    string ssn;
    double netPay;
};
Derived class: HourlyEmployee

class HourlyEmployee : public Employee {
// Instantly inherits all member functions and variables of class 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( ); // plan to redefine printCheck function
private:
  double wageRate;
  double hours;
};
Writing derived classes

- 3 possibilities for member functions:
  - Inherit – i.e., do nothing
  - Redefine – have new method act differently
  - Define new – add abilities not in base class at all
- 2 possibilities for member variables:
  - Inherit – though if private, may not directly access/set
  - Define new – more data in addition to base class data
- Notice: cannot redefine member variables – attempts to do so will create "shadow variables"
  - i.e., just creates a new variable with the same name, effectively hiding the inherited one – usually a mistake
Derived class constructors

- A base class constructor is *always* invoked first
  - i.e., first task of derived class constructor's initialization list
  - If not done explicitly, base class default constructor will be called implicitly
    - Will result in compile error if base class has no default ctor
- Need explicit call to use an alternative base class ctor
  - Syntax: `BaseClassName(arg1, arg2, ...)`
- Derived Employee example:
  ```cpp
  HourlyEmployee::HourlyEmployee(string name, string number, double rate, double hours)
    : Employee(name, number), wageRate(rate), hours(hours) 
  { } 
  ```
A subclass object's composition

- **Remember**: a derived class definition just defines part of the resulting object
  - The rest of the object is the base class portion
Redefining ≠ overloading

- Redefining only applies to a derived class
  - Same parameter list (i.e., same "signature")
  - Essentially "re-writes" the same function
- Overloading can happen in base or derived
  - Different parameter list – different signature
  - Defining a new function with the same name
- Recall definition of a signature:
  - Name(parameter list)
  - Does not include return type, and ' &' ignored
Accessing redefined base function

- A redefined base class definition is not "lost"
  
  Employee jane;
  HourlyEmployee sally;
  jane.printCheck(); // Employee function
  sally.printCheck(); // HourlyEmployee function
  sally.Employee::printCheck();
  // uses scope resolution to call Employee function!

- Often done while implementing derived class – let base function do some of the work
Some functions are not inherited

- All "normal" functions in the base class are inherited in the derived class
- The exceptions ("abnormal" functions?):
  - Constructors and destructor
  - Copy constructor and assignment operator
- Compiler generates default versions if you don't redefine them in the derived class
  - But remember that can be problematic if pointing to dynamic memory, so often should redefine
Subclass operator= and copy ctor

- Although not inherited, a derived class typically must use the base class’s versions
- e.g., an operator= in class D : public B
  
  ```cpp
  D& D::operator=(const D &right) {
    // first call assignment operator of base class to take
    // care of all the inherited member variables
    B::operator=(right);
    ... // then set new variables of derived class
  }
  ```

- Copy ctor must use base class version too
  ```cpp
  D::D(const D &other) : B(other), ... { }
  ```
Destructors in derived classes

- Easy to write if base class dtor is correct
  - No need to call base class dtor – because it is called automatically at the end of the derived class’s dtor

- So derived class destructors need only worry about derived class variables
  - Usual purpose: release resources allocated during the object's life
  - Let base class dtor handle inherited resources
Examples: PFArrayD and ...Bak

● **Base class** `PFArrayD`:
  – Stores a *pointer* to a double array on free store
    ● Array has a fixed capacity after construction
  – Has mgr., other functions, plus `[]` and `=` ops

● **Derived class** `PFArrayDBak`:
  – Has pointer to its *own array* – can be used to backup and restore data in base class's array
  – Redefines ctors, dtor and operator=
Writing derivable classes

- Always provide a constructor that can be called with no arguments
- Control subclass' access to member variables and functions as appropriate – three choices:
  - public members are accessible to all other classes
  - private members are not directly accessible to any other class – should be used for most variables, and also appropriate for "helper" functions
  - A third choice is protected member access
    - Only subclasses (those derived from this one) can access
    - Some consider it bad OOP practice – violates info hiding
**protected/private inheritance**

- **Note:** rarely used; frankly a little weird
  - Destroys “is a” relation of derived class object
- **Protected inheritance** – all public members in the base class become protected members in the derived class
  ```csharp
class SalariedEmployee : protected Employee {...}
```
- **Private inheritance** – all members in the base class become private in the derived class
  ```csharp
class SalariedEmployee : private Employee {...}
```
Many more inheritance issues

● For instance: Sometimes it is better to use “has a” instead of “is a” relationship
  – Means one class has an object of another class
  – Generally a more flexible design

● Can also do multiple inheritance in C++
  class ClockRadio :
    public Radio, public AlarmClock;
  – Tricky though (more later, after virtual keyword)

● “Slicing” and “upcasts” – more to come
Virtual functions – concepts

- **Virtual**: exists in essence though not in fact
- Idea is that a virtual function can be “used” before it is defined
  - And it might be defined many, many ways!
- Relates to OOP concept of **polymorphism**
  - Associate many meanings to one function
- Implemented by **dynamic binding**
  - A.k.a. late binding – happens at run-time
Polymorphism example: figures

- Imagine classes for several kinds of figures
  - Rectangles, circles, and ovals (to start)
  - All derive from one base class: Figure

- All “Figure” objects inherit: void draw()
  - Of course, each one implements it differently!

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

- Nothing new here yet …
Figures example cont. – center()

- Consider that base class Figure has functions that apply to “all” figures
- e.g., center(): moves figure to screen center
  - Erases existing drawing, then re-draws the figure
  - So Figure::center() uses draw() to re-draw
- But which draw() function will be used?
  - We’re implementing base class center() function, so we have to use the base class draw() function. Right?
- Actually, it turns out the answer depends on how draw() is handled in the base class
Poor solution: base works hard

- Figure class tries to implement draw to work for all (known) figures
  - First devise a way to identify a figure’s “type”
  - Then Figure::draw() uses conditional logic:
    ```cpp
    if ( /* the Figure is a Rectangle */ )
        Rectangle::draw();
    else if ( /* the Figure is a Circle */ )
        Circle::draw();
    ...
    ```

- But what if a new kind of figure comes along?
  - e.g., how to handle a derived class Triangle?
Better solution: virtual function

- Base class declares that the function is virtual:
  
  ```cpp
  virtual void draw() const;
  ```

- Remember it means `draw()` exists in essence

- Such a declaration tells compiler “I don’t know how this function is implemented, so wait until it is used in a program, and then get its implementation from the object `instance`.”

- The instance will exist in fact (eventually)
  - Therefore, so will the implementation at that time!

- Function “binding” happens late – dynamically
Another virtual function example

- Record-keeping system for auto parts store
  - Track sales, compute daily gross, other stats
  - All based on data from individual bills of sale
- Problem: lots of different types of bills
- Idea – start with a very general Sale class that has a virtual bill() function:
  ```cpp
  virtual double bill() const;
  ```
- Rest of idea – many different types of sales will be added later, and each type will have its own version of the bill() function
Sale functions: savings and op <

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 functions use member function bill()!
A class derived from Sale

class DiscountSale : public Sale {

public:
    DiscountSale();
    DiscountSale(double price, 
                 double discount);
    double getDiscount() const;
    void setDiscount(double newDiscount);
    double bill() const; // implicitly virtual

private:
    double discount;   // inherits price
};
DiscountSale’s bill() function

- First note – it is automatically virtual
  - Inherited trait, applies to any descendants
  - Also note – rude not to declare it explicitly
- Of course, definition never says virtual:
  ```cpp
double DiscountSale::bill() const {
    double fraction = discount/100;
    return (1 - fraction)*getPrice();
}
```
  - Must use access method as price is private
The power of virtual is actual!

- e.g., base class Sale written long before derived class DiscountSale
- Sale had members savings and ‘<’ before there was any idea of class DiscountSale
- Yet consider what the following code does
  
  DiscountSale d1, d2;
  d1.savings(d2); // calls Sale’s savings function

- In turn, class Sale’s savings function uses class DiscountSale’s bill function.

  Wow!
Clarifying some terminology

- Recall that overloading ≠ redefining
- Now a new term – **overriding** means *redefining a virtual function*
- Polymorphism is an OOP concept
  - Overriding gives many meanings to one name
- Dynamic binding is what makes it all work
- “Thus,” as Savitch puts it, “polymorphism, late binding, and virtual functions are really all the same topic.”
Why not all virtual functions?

- Philosophy issue: pure OOP vs. efficiency
  - All functions are virtual by default in another popular programming language (Java) – there must take steps to make functions non-virtual
  - C++ default is non-virtual – programmer must explicitly declare (except when inherited trait)
- Virtual functions have more “overhead”
  - More storage – for class virtual function table
  - Slower – a look-up step; less optimization