Inheritance (with C++)

Starting to cover Savitch Chap. 15

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)

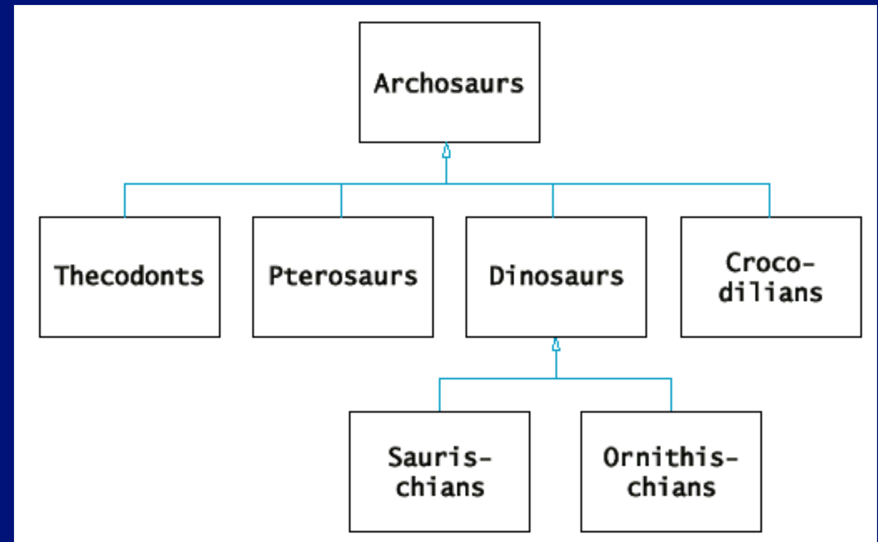
Inheritance begets hierarchies

- "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 …

Base class example: Employee

```
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 methods and data 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;  // new data specific to this derived class
    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

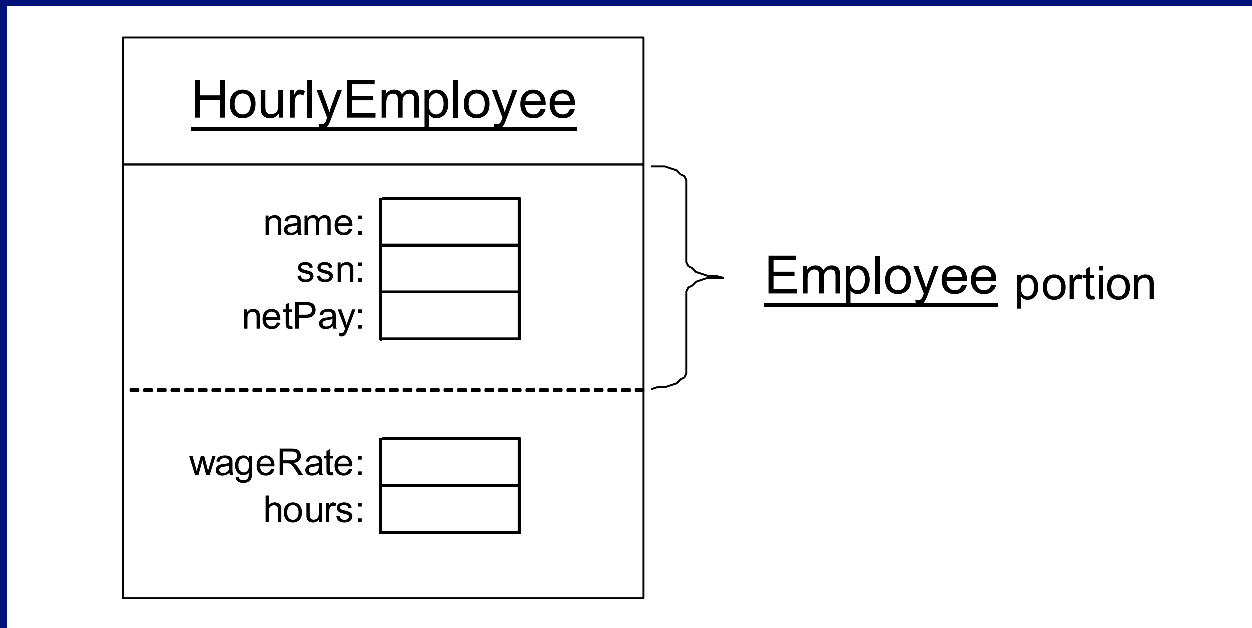
- A base class constructor is *always* invoked first
  - i.e., first task of derived class constructor's initialization list
  - If no explicit call, base class default constructor will be called implicitly (compile error if base class has no default ctor)
- Must explicitly call to use an alternative base class ctor
  - Syntax: `BaseClassName(arg1, arg2, …)`
- Derived Employee example:

```
HourlyEmployee::HourlyEmployee(string name,
    string number, double rate, double hours)
    : Employee(name, number), wageRate(rate),
      hours(hours)
{ }
```

  - Properly *initializes* `name`, `ssn`: private Employee data

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 implmenting 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
  - 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`

```
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

```
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

~mikec/cs32/demos/
SavitchAbsolute_ch14/
PFArrayD.h

- 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

…PFArrayDBak

- 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

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

- Private inheritance – all members in the base class become private in the derived class

```
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
- First an application: simulating an ecosystem
  - organism.h and pondlife.cxx (.../demos/ecosystem/)