CSE 143

Class Relationships and Inheritance

[Chapter 8, pp.343-354]

"Class Relationships"

- ... is the title of Chapter 8
- Let's step back first to why we care
  - Computer programs make a model of the world in order to solve some problem
  - In the world, there are objects, and there often are relationships between them

One Kind of Relationship

- At the bank:
  - Savings accounts and checking accounts are two *kinds of* accounts
  - Commercial customers and regular customers are two *types of* customers
  - Tellers, loan officers, and branch managers are *kinds of* employees
  - Customers and employees are kinds of people
- "Kind of" relationships are everywhere
  - and therefore very important in modeling the real world in a computer

Has-a vs. Is-a

- A pencil has an eraser
  - But an eraser is not a kind of pencil or vice-versa
- A pencil is a kind of writing tool
  - But a pencil doesn't contain a writing tool or vice-versa
- lions/tigers/teeth/mammals
  - What relationships?

Class in a class: "has-a"

- Use an instance of one class as a member variable in another

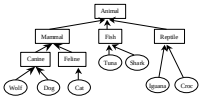
```
class StudentCouncil {
    Student president;
    Student minister_of_propaganda;
    Student members_at_large[5];
};
```

- A "has-a" relationship
  - A `StudentCouncil` "has-a" `Student`

    `In fact, more than one`
- We've used this plenty already (e.g., strings, …)
- For "is-a", we'll need a different C++ feature

Hierarchies of Organization

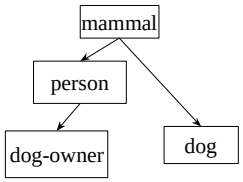
- Often, we classify things in a hierarchy from general to specific



- Hierarchies show "is-a-kind-of" relationships
  - A Dog "is-a-kind-of" Canine, a Shark "is-a-kind-of" Animal
  - A Stack "is-a-kind-of" OrderedCollection

Caution: Not Every Relationship is "is-a" or "has-a"

- dog/dog-owner



owner *walks* dog

owner *feeds* dog

dog *bites* owner

Is-a instance vs Is-a kind of

- Commercial Customer is a kind of Customer
  - Both are types
- Instances of types are by now a very familiar programming concept
- One type being a kind of another type is a new concept
- Compare "Fluffy is a cat" vs. "Cats are carnivores."

Why Focus on "is-a" and "has-a"?

- A way to take advantage of redundancy
- If Appointment contains ("has-a") Date, and Date is already defined, we don't have to start from scratch
  - C++: use one type inside another
  - Have seen lots of examples already
- "Is-a kind of" would be another way to take advantage of redundancy
  - If I had Mammal defined, a lot of that would carry over to Lion.
  - For "is-a", we need some new C++ stuff: *inheritance*

Modeling a Bank

- Bank has name
- Has branches
- Branches have customers
- Customers have accounts
- Multiple kinds of accounts (savings, checking, etc).
- Multiple kinds of people (employees vs customers)
  - Multiple kinds of employees (tellers, loan officers, VPs, etc.)

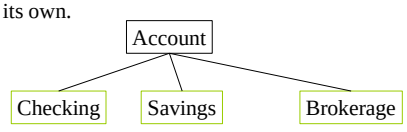
Object - Bank Account

- Accounts have certain data and operations
  - Regardless of whether checking, savings, etc.
- Data
  - account number
  - balance
  - owner
- Operations
  - open
  - close
  - get balance
  - deposit
  - withdraw

Kinds of Bank Accounts

Checking	Savings	Brokerage
Monthly fees	Interest rate	List of stocks
Minimum bal.		and bonds

Each type shares some data and operations of "account", and has some data and operations of its own.



Inheritance in C++

```
class Account {
...
  double balance;
  Customer owner;
  Date dataOpened;
...
   void makeDeposit (double
     Amount);
...
};
```

```
class SavingsAccount : public
  Account  {
...


... double interestRate;
...
   ...
   void creditInterest( );

};
```


A Big Idea

- Inheritance is a BIG IDEA
- One of the great new features of C++
- A key concept in modern programming
- Essential for using today's languages, tools, and libraries
- However...
  - The details in C++ can get messy
  - Sometimes very, very, very, very messy.

Toward Object-Oriented Programming

- **Inheritance** is a major aspect of what is called "object-oriented programming".
- Another is **encapsulation**, which we're already using.
  - Data and methods packaged together in classes
  - Public/private access control
- A third is **polymorphism**
  - Constructor overloading is one example
  - Operator overloading is another example
  - We'll also see virtual functions
- Finally, OO is a matter of world-view rather than just programming techniques

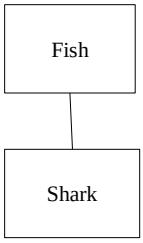
Inheritance Terminology

- Inheritance is a way to encode the "is-a-kind-of" relation in OO languages
  - Shark declares that it "is-a-kind-of" Fish by inheriting from Fish
- A *derived class* inherits from a *base class* by putting `: public BaseClassName` in the class declaration

derived class (or subclass) → **base class** (or superclass)

```
class Shark : public Fish {
    // Shark-specific stuff here
};
```


Picturing the Hierarchy



base class (or superclass)

derived class (or subclass)

All data and methods in base class (superclass) are automatically inherited by derived (sub) class

Example: A `Point` Class

- We're building a graphics system...
- Let's say we had the following class "point"

```
class Point {
public:
    Point( double x, double y );

    double getX();
    double getY();

    void print( ostream& os );

private:
    double xpos;
    double ypos;
};
```

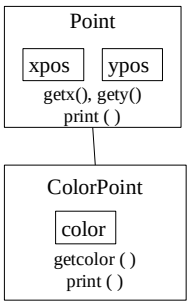
- We can use inheritance to create a class of colored points based on this class

`ColorPoint` Via Inheritance

- `ColorPoint` "is-a" `Point`
- Therefore `ColorPoint` has to be able to do anything `Point` can
- All fields and methods of `Point` are "inherited" by `ColorPoint` - they are transparently included!
- Derived class can add new methods, fields
- Derived class can override base class behavior (methods)

```
class ColorPoint : public Point {
public:
    ColorPoint( double x, double y,
                Color c );

    // getX() is inherited from Point
    // getY() is inherited from Point

    // New accessor method for the
    // Color field
    Color getColor();

    // We still need to redefine
    // the print method!
    void print( ostream& os );

private:
    // xpos is inherited from Point
    // ypos is inherited from Point
    Color color;
};
```


Point Hierarchy



base class
(or superclass)

derived class
(or subclass)

Rules of Inheritance

- All data and methods in base class (superclass) are automatically inherited by derived (sub) class
  - Changes in base class are automatically propagated into derived classes
- What about the print( ), which exists in both?
  - The derived version overrides
- What if you wanted to override xpos and ypos?
  - Sorry, not allowed
- So ColorPoint inherits xpos and ypos
  - Problem: xpos and ypos are private, right? Need some more rules....

Public/Private/Protected

- **Public** members of base class: visible to derived class and clients that use it
- **Private** members of base class: still not visible to derived class or clients
  - The private members are still there inside the derived object! They just aren't visible
- Protected members in base class: visible in derived class, but not visible to clients.
- Advice: When in doubt, use "protected" (maybe)
  - If you expect the current class to be extended later
  - If you don't mind exposing implementation details to subclasses

ColorPoint Implementation

```
Color ColorPoint::getColor(){
    return color;
}


void ColorPoint::print ( ostream& os ){
    os << "(" << getX() << ", " << getY()
       << ")/" << color;
}
```


ColorPoint Constructor

```
ColorPoint::ColorPoint( double x, double y, Color c )
    : Point( x, y ){
    color = c;
}
```

- New notation: *": baseclass(args, …)"* calls base class constructor
  - will initialize base class fields in derived class object
  - Must be placed here
    Can't call directly inside the function
- This "initializer" list can also call constructors for member variables

Inheritance and Constructors

- Constructors are not inherited!
  - Can't be, because their name specifies which class they're part of!
- Review: Constructors are called in "inside-out" order
- Constructor of base class is called before constructor of derived class executes
  - Explicitly: *":class(arguments)"* in initializer list
  - Automatically: If explicit call omitted, default constructor of base class is called

`ColorPoint` Client

```
Point p( 1.0, 0.0 );
ColorPoint cp1( 3.14, -45.5, RED );

cp1.print( cout );
// No problem: ColorPoint::print is defined

p.print( cout );
// No problem: Point::print is defined

cout << cp1.getX() << " " << cp1.getY() << endl;
// No problem: calls Point::getX() and Point::getY()
// on Point subset of ColorPoint to access private
// xpos and ypos fields

 .... p.getColor (); ...
      // OK or not?
```


Substituting

```
Point p( 1.0, 0.0 );
ColorPoint cp1( 3.14, -45.5, RED );
```

General rule (memorize): *an instance of a derived class can always be substituted for an instance of a base class*

Derived class guaranteed to have (at least) the same data and interface as base class

"If it's true of a mammal, it's true of a dog"

Footnote: Invoking Overriden Methods

- What if I really want to call Point's print method, rather than ColorPoint's version of it?
  - `Might want to do this to reuse code`
- What happens if we try it as follows?

```
void ColorPoint::print( ostream& os ){
    print( os );
      // intending to call print method in superclass
    os << ", " << Color;
}
```


Solution: Class Scope Resolution Operator

- It turns out that the :: operator allows us to explicitly call an overriden method from the derived class

```
void ColorPoint::print( ostream& os ){
    Point::print( os );
    os << ", " << Color;
}
```

- `BaseClass::method( arguments )` can be used as long as `BaseClass` really is a parent class (either direct base class or more distant ancestor)

Draw the Hierarchy

```
//assume all methods are public

class animal {...
  dance ( );
... };

class mammal : public
  animal {...
  dance ( );
  walk ( );
...};
```

```
class hedgehog : public
  mammal {...
        // no "dance" method
        dig ( );
        walk ( );
        walk (int, int);
...};

class seaUrchin : public
  animal {...
        dance ( );
        sting ( );
};
```

What's Legal / Which function is called?

- hedgehog harry;
- seaUrchin ursula;
- mammal mona;
- harry.dance ();
- ursula.dance();
- mona.dance();
- harry.walk ();
- harry.walk (1 , 0);
- ursula.walk ();
- mona.walk ();