CSE 143

Object-Oriented Programming

[Chapter 8]

Introduction

- Classes have given us tremendous benefit
  - Encapsulation: make details of implementation and representation private
  - Classes help break down and organize the task of writing a program
- Classes are designed to enable other benefits
  - Inheritance (subclassing)
  - Dynamic dispatch (polymorphism)
- Two very powerful programming tools!
  - They help us write less code

Classes Using Other Classes

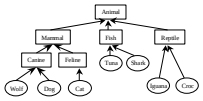
- We've seen one way that a class can make use of another class
  - Use an instance of that class as a member variable

```
class PolyLine
{
 private:
    PointArray pa;
};
```

- We might call this a "has-a" relationship
  - A **`PolyLine`** "has-a" **`PointArray`**

Hierarchies of Organization

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



- Objects have more of a "is-a-kind-of" relationship
  - A Dog "is-a-kind-of" Canine, a Shark "is-a-kind-of" Animal
  - This is often just called a "is-a" relation

Inheritance

- 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 it
  - 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
};
```


Example: A **`Point`** Class

- Let's say we had the following class
- We can use inheritance to create a class of colored points based on this class

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

    double getX();
    double getY();

    void print( ostream& os );

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

`ColorPoint` with 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 behaviour

```
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;
};
```


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
- 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
- Can also declare “protected” members in base class which are visible in derived class, but not visible to clients.

ColorPoint Implementation

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

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

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

- New notation: “*: member(values, …)*” specifies base class constructor to initialize base class fields in derived class object

`ColorPoint` Client

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

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

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

// p can refer to any Point, including one that is
// actually a ColorPoint.  But p can only be used to
// access fields that belong to all Points (i.e.,
// can't be used to access getColor
p = cp1;
```


Invoking Overriden Methods

- Observation: **`ColorPoint::print`** does the same thing as **`Point::print`**, and then prints out a Color
- So perhaps we can call **`Point::print`** from within **`ColorPoint::print`**
- What happens if we try it this way?

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


Scope Resolution

- 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)

Inheritance and Constructors

- Constructors are not inherited!
  - Can't be, because their name specifies which class they're part of!
- Instead, constructor of base class is called automatically before constructor of derived class
  - Can indicate base class constructor explicitly using the *":class(arguments)"* notation to pass parameters (like in **`ColoredPoint`** example)
  - If omitted, default constructor of base class is called
  - Constructors are called in "inside-out" order

Substituting Derived Classes

- Recall that 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
- But you may not get the behaviour you want!

```
Point p( 1.0, 9.0 );
ColorPoint cp( 6.0, 7.0, red );

p = cp;

void printPoint( Point pt )
{
    pt.print( cout );
}

printPoint( p );
printPoint( cp );
```


Pointers And Inheritance

- You can also substitute a *pointer* to a derived class for a *pointer* to a base class
  - There's still that guarantee about data and interface
  - Also holds for reference types
  - **No information disappears!!**
- Unfortunately, we still have the same problems…

```
Point *pptr = new Point( 1.0, 9.0 );
ColorPoint *cpptr =
    new ColorPoint( 6.0, 7.0, red );
Point *fooptr = cpptr;

void printPoint( Point *ptr )
{
    ofstream ofs( "point.out" );
    ptr->print( ofs );
    ofs.close();
}

printPoint( pptr );
printPoint( cpptr );
```


Static And Dynamic Types

- In C++, every variable has a static and a dynamic type
  - Static type is declared type of variable

    `Every variable has a single static type that never changes`
  - Dynamic type is type of object the variable actually contains or refers to

    `Dynamic type can change during the program!`
- Up to now, these have always been identical
  - But not any more!

```
Point *myPointPointer = new ColorPoint( 3.14, 2.78, green );
```


Static Dispatch

- "Dispatching" is the act of deciding which piece of code to execute when a method is called
- Static dispatch means that the decision is made statically, *i.e.* at compile time
  - Decision made based on static type of receiver

```
Point *myPointPointer = new ColorPoint( 3.14, 2.78, green );

// myPointPointer is a Point*, so call Point::print
myPointPointer->print( cout );
```

  - Idea: make the decision at runtime, based on the *dynamic* type of the object

Dynamic Dispatch

- C++ has a mechanism for declaring individual methods as dynamically dispatched
  - If an overriding function exists, call it
- In base class, label the function with **`virtual`** keyword
  - Overriding versions in subclasses may or may not have the **`virtual`** keyword; but use consistently for better style
  - Rule of thumb: If you may ever need to override a method, make it **`virtual`**!
  - Even safer rule: Unless you have a good reason not to, member functions in a class hierarchy should be virtual.

Example Of Dynamic Dispatch

```
class Point {
public:
    virtual void print( ostream& os );
    …
};

class ColorPoint : public Point {
public:
    virtual void print( ostream& os );
    …
};

Point *p = new ColorPoint( 3.13, 5.66, ochre );

p->print( cout ); // calls ColorPoint::print( )
```


Dynamically-Dispatched Calls

```
Point *p = new ColorPoint( 3.13, 5.66, ochre );
p->print( cout );
```

- The compiler notices that **`Point::print`** is defined as **`virtual`**
- Instead of just calling **`Point::print`**, it inserts extra code to look at information attached to the object by **`new`** to decide what function to call
- This is slightly slower than static dispatch
  - Almost always too minor a speed penalty to worry about

A User-Interface Example

- Many user interface toolkits are implemented as class hierarchies
- Helps manage complexity
- Question: how should we implement **`Widget::draw`**?

```
class Widget {
public:
    void getPosition( int& x, int& y );
    virtual void draw( Screen& theScreen );
    …
private:
    int xpos;
    int ypos;
};

class Button : public Widget {
public:
    virtual void draw( Screen& theScreen );
    …
};

class Toolbar : public Widget {
public:
    virtual void draw( Screen& theScreen );
};
```


Abstract Classes

- Some classes are so abstract that instances of them shouldn't even exist
  - What does it mean to have an instance of **`Widget`**? of **`Animal`**?
- An *abstract class* is one that should not or can not be instantiated - it only defines an interface
- A *concrete class* can have instances
- It may not make sense to attempt to fully implement all functions in an abstract class
  - What should **`Widget::draw`** do?

Pure Virtual Functions

- A "pure virtual" function is not implemented in the base class - must implement in derived class
- Syntax: append "**`= 0`**" to base method declaration

```
class Widget {
public:
    virtual void draw( Screen& theScreen ) = 0;
};
```

- Compiler guarantees that class with pure virtual functions cannot be instantiated
- If you call a pure virtual function, you'll use the version from some derived class

```
Widget *w = new Button();
w->draw( myScreen );
```


Summary

- Object-Oriented Programming
- Inheritance
  - Use for "is-a-kind-of", not "has-a" relations
  - Classification hierarchies
  - Base class (superclass), derived class (subclass)
  - Overriding functions
- Static and dynamic types
- Dynamic Dispatch
  - Virtual functions
  - Abstract classes, pure virtual functions