Object-Oriented Programming
[Chapter 11]

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

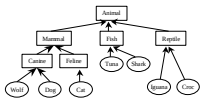
- So far, we've seen one way that a class can make use of another class
  - Use an instance of that class as a member variable

```
class StudentCouncil
{
    Student president;
    Student minister_of_propaganda;
};
```

- We might call this a "has-a" relationship
  - A **`StudentCouncil`** "has-a" **`Student`**

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
  - A Stack "is-a-kind-of" OrderedCollection

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)

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

class List : public SequentialCollection {
    …
};
```

base class (or superclass)

Example: A **`Point`** Class

- Let's say we had the following class
- We can use inheritance to store a colour field for points without changing this declaration!

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

    double getX();
    double getY();

    void print( ostream& os );

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

`ColourPoint` Without Inheritance

- This isn't inheritance
- This is the interface we want, but we'll have to redefine all `Point` methods
- Inheritance will let us write less code!

```
class ColourPoint
{
public:
    ColourPoint( double x, double y,
                 Colour c );

    double getX();
    double getY();

    void print( ostream& os );

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


`ColourPoint` With Inheritance

- `ColourPoint` "is-a-kind-of" `Point`
- Therefore `ColourPoint` has to be able to do anything `Point` can
- All fields and methods of `Point` are "inherited" by `ColourPoint` - they are transparently included!
- Subclass can add new methods, fields
- Subclass can override superclass behaviour

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

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

    // New accessor method for the
    // colour field
    Colour getColour();

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

private:
    // xpos is inherited from Point
    // ypos is inherited from Point
    Colour colour;
};
```


Rules of Inheritance

- All data and methods in superclass are automatically inherited by subclass
  - As if you copied them info the subclass yourself
- Changes in superclass are automatically propagated into subclasses
- Public members of superclass visible to subclass and client of subclass
- Private members of superclass still not visible to subclass or its clients

ColourPoint Implementation

```
ColourPoint::ColourPoint( double x, double y, Colour c )
    : Point( x, y )
    , colour( c )
{}

Colour ColourPoint::getColour()
{
    return colour;
}

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

- Superclass is initialized using memberwise initialization on superclass name

`ColourPoint` Client

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

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

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

// Assign p to be the "Point slice" of cp1, the part
// of cp1's representation which was inherited from
// Point
p = cp1;
```


Subclass Tips

- Add "`: public BaseClassName`" to the first line of class declaration
- Add "`: BaseClassName(arguments)`" to beginning of subclass constructor's definition
  - Only needed to invoke non-default constructor
  - Default constructor of base class is otherwise invoked automatically
- Don't rewrite inherited methods, instead inherit them!
- Don't change or re-declare inherited data members

When to Use Inheritance?

- Most appropriate if all of the following are true:
  - The subclass A "is-a-kind-of" the base class B
  - Interface of derived class is superset of base class interface (all functions from base and then some)
  - Representation of derived class is superset of base class representation (all data from base and then some)
  - Subclass implementation is similar to implementation of base class

When Not to Use Inheritance?

- Class **`B`** should not inherit from class **`A`** when
  - Derived class has the base class as a part
    - When **`B`** "has-a" **`A`**, or when **`A`** "is-part-of" **`B`**
    - A **`bird`** has-a **`wing`**, but is not a-kind-of **`wing`**
  - Derived class doesn't have same representation and/or interface as base class
    - Can often reorganize class hierarchy to fix the problem
    - Use different class as base class
    - Make small changes to other classes

Subclasses And Private Data

- It seems unfair to restrict subclass access in the same way as client acces
  - Subclasses are a kind of "priveleged" client of the superclass
- Sometimes, the superclass would like to give subclass access, but not clients
  - Public is too open, private is too closed
  - Need third access specifier!

Protected Access

- In addition to **`public:`** and **`private:`**, there is a **`protected:`** access specifier
  - Visible to original class, visible in subclasses, still hidden from clients
  - Class declaration can have any number of public, protected and private sections

Example Of Protected Access

```
class Point {
public:
    …
protected:
    double xpos;
    double ypos;
};

class ColourPoint : public Point {
public:
    void setTemperature( double deg );
    …
protected:
    Colour colour;
};

void ColourPoint::setTemperature( double deg )
{
    colour = setColourFromTemperature( deg );
    ypos = (deg + 40.0) / 150.0; // OK now!
}
```


Invoking Overriden Methods

- Observation: **`ColourPoint::print`** does the same thing as **`Point::print`**, and then prints out a colour
- So perhaps we can call **`Point::print`** from within **`ColourPoint::print`**
- What happens here?

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

Scope Resolution To The Rescue!

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

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

- **`BaseClass::method( arguments )`** can be used as long as **`BaseClass`** really is a superclass
- Remember that superclasses are "transitive"

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 call base class constructor explicitly to pass parameters (like in **`ColouredPoint`** 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 probably don't get the behaviour you want!

```
Point p( 1.0, 9.0 );
ColourPoint 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 );
ColourPoint *cpptr =
    new ColourPoint( 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
    - Dynamic type can change during the program!
- Up to now, these have always been identical
  - But not any more!

```
Point *myPointPointer = new ColourPoint( 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 ColourPoint( 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 receiver!

Dynamic Dispatch

- C++ has a mechanism for declaring individual methods as dynamically dispatched
  - "Don't necessarily call this function; call the overriding version, if it exists"
- In base class, method is labeled 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`**!

Example Of Dynamic Dispatch

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

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

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

p->print( cout ); // It's alive!  ALIVE!!!
```


Another Example Of Dynamic Dispatch

```
class Gizmo {
public:
    virtual bool moveRay(Ray& r);
    …
};

class FslashGizmo : public Gizmo {
public:
    virtual bool moveRay( Ray& r );
    …
};

…
Gizmo **data = new Gizmo*[ 100 ];
data[ 13 ] = new FslashGizmo();
data[ 14 ] = new Gizmo();
…
data[ 13 ]->moveRay( ray );
data[ 14 ]->moveRay( ray );
…
```


How Does It Work?

- It's a secret!

How Does It Work!?!!

```
class Point {
public:
    Point( double x, double y );
    virtual void print( ostream& os );
private:
    double xpos;
    double ypos;
};

int main( void )
{
    cout << sizeof( Point ) << endl;
    cout << sizeof( double ) + sizeof( double ) << endl;
    return 0;
}
```

- This program prints out 24 and 16!
- The compiler is automatically inserting extra information into the class: **`__vfptr`**

Dynamically-Dispatched Method Calls

```
Point *p = new ColourPoint( 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 inside **`Point`**'s **`__vfptr`** to decide what function to call
- This is slightly slower than static dispatch
  - Major research area in programming languages
  - Probably not so much slower that you should worry about it

A Practical 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 **`Collection`**? of **`Animal`**?
- An *abstract class* is a class that should not or can not be instantiated
  - A *concrete class* can have instances
- Not every abstract class can be completely implemented
  - What would we write for **`Widget::draw`**?

Pure Virtual Functions

- A pure virtual function is a placeholder that concrete subclasses must fill in
- Syntax: append "**`= 0`**" to method declaration

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

- Compiler guarantees that class with pure virtual functions cannot be instantiated
- But you can still call them!

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


Summary

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